Debugging With Behavioral Watchpoints

Akshay Kumar

October 6, 2013

Abstract

Finding, understanding, and fixing bugs in software systems 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 to instrumentation tools, making it hard to implement such tools.

In this theis, we introduce *behavioral watchpoints*, a new software-based watchpoint framework that simplifies the implementation of DBT-based program analysis and debugging tools. Behavioral watchpoints have two key features: 1) they provide contextual information at the instruction level which are directly available with watchpoints and 2) they enable specializing instruction-level instrumentation with individual data structures. We describe three applications that were easily developed using our watchpoint framework: detecting buffer overflows, detecting read-before-write and memory freeing bugs and detecting memory leaks. We implemented behavioral watchpoints using Granary, a DBT framework for instrumenting operating system kernels. We evaluated the overheads of watchpoints for analyzing and debugging operating system kernel modules and show that these overheads are reasonable.

Contents

List of Tables

List of Figures

Chapter 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 the corrupt data. Such an instruction may be difficult or prohibitively costly to find. The advent of multicore systems and the ever-increasing size and complexity of software has made debugging even more challenging. This makes it important to develop tools that can handle large and more complex programs. At the same time there is a need for tools to provide interesting information about program execution that can be used to improve the quality of the programs.

Dynamic binary translation (DBT) systems provide a powerful infrastructure for building program debugging and analysis tools. A DBT system enables monitoring and potentially manipulating every instruction in an existing binary before it executes, helping detect programming errors, and thus improving program dependability. Instruction-grained inspection can help detect the most subtle program faults including low level issues such as memory bugs [30] and security errors [31, 12], as well as high level issues such as concurrency errors [34, 1] and interface errors in multilingual programs [22]. Several popular DBT frameworks, such as DynamoRIO [6],

Pin [7] and Valgrind [30] have been used develop such debugging tools. Existing tools such as Memcheck [41] and Helgrind [26] are developed over Valgrind which uses 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. Other tools like Program shepherding [21] and vx32 [15] use DBT to improve program security and enforce modularity. The use of DBT system in developing these tools has three distinct advantages.

i) The DBT system operates at instruction granularity giving the tools complete control over program execution. It provides the tools the ability to observe and modify any executed instructions and gather any runtime information.

ii) The DBT system operates on binaries and does not require the client program to be changed or compiled in any particular way, making it easy to use.

iii) The DBT system naturally allows instrumenting all code. Instrumenting all client code statically can be difficult if code and data are mixed or different modules are used, and is impossible if the client uses dynamically generated code. The ability to instrument all code is crucial for correct and complete handling of third-party code such as pre-compiled libraries.

These advantages make dynamic binary instrumentation (DBI) a compelling technique for developing many dynamic analysis tools. However, using a DBT framework for developing these powerful and efficient debugging applications is challenging for three reasons.

First, a DBT system provides infrastructure for instrumenting program code, an approach we call 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. The code-centric approach makes it harder to implement these debugging tools.

Second, DBT abstractions are too low-level: individual instructions only reveal what memory addresses are being accessed. This is at odds with specialising instruction-level instrumentation

with higher-level abstractions. For example, developing a tool to debug a data corruption problem in a specific field of a data structure (e.g., `i_flags` field of an `inode` structure of Linux file system), or detecting an invariant violation in a data structure requires instrumentation to be specialised 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 `inode`s.

In this work, 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 [48], it supports 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.

The other 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 watchpoint 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` 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 allowing comprehensive instrumentation of module code and on-demand translation of kernel code.

We have implemented behavioral watchpoints using Granary, a DBT framework designed to instrument kernel modules [16, 14]. Granary instruments arbitrary, binary Linux kernel modules efficiently and without imposing overhead when the core kernel is running and provides a powerful infrastructure for debugging and analysis of kernel modules. We have prototyped several module debugging applications using behavioral watchpoints. These applications include a buffer-overflow detector, a memory leak detector, and a shadow-memory based tool for logging the access patterns of different classes of modules for detecting buggy or malicious behavior.

1.1 Thesis contributions

This thesis makes the following contributions:

1. We describe the design and implementation of the behavioral watchpoint framework. The framework is implemented using Granary, a DBT system for kernel modules, which provides access to the contextual-information on every memory access. We also describe code-centric and trap-based approaches for implementing behavioral watchpoints.

2. We evaluate the two approaches by comparing the effort involved in developing tools and the performance overheads of the two approaches. We discuss the different use cases of both the approaches in debugging operating system kernels.

3. We also describe the prototype applications that we developed using behavioral watchpoints and have used for debugging and analysis of some kernel modules. We developed three

different applications, the buffer-overflow detector, selective shadowing and leak detector tool for debugging memory issues in kernel modules.

The rest of the thesis is structured as follows. Chapter 2 describes Granary and its various features that are helpful in developing the behavioral watchpoint framework. Chapter 3 describes our design and approach for implementing behavioral watchpoints. Chapter 4 discusses the several prototype applications that we developed using watchpoints. Chapter 5 discusses the related work closely associated with the field. Finally, Chapter 6 concludes the thesis and highlights directions for future work

Chapter 2

Granary

This chapter provides background information about Granary [16], on which we build our watchpoints. Granary is a dynamic binary translation (DBT) framework designed to instrument kernel modules, which are a frequent source of bugs and vulnerabilities in operating systems [8, 24].

Granary instruments arbitrary, binary Linux kernel modules efficiently and without imposing overhead when the core kernel is running. Granary is unique among DBT frameworks 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. Granary is designed with three goals for practical module analysis: i) comprehensively analyze *all* modules; ii) impose no performance overheads on non-module kernel code; iii) require no changes to modules and minimal changes to the kernel, so that it can be easily ported between different hardware and kernel versions. Granary meets all these goals as follows:

1. Granary is comprehensive because it controls and instruments the execution of all module

code. Granary maintains control by ensuring that normal module code is never executed. Instead, only decoded and translated module code is executed. Translated module code contains instrumentation and always yields control back to Granary. The ability to comprehensively instrument the kernel modules help enables implementing watchpoints on an arbitrary memory reference.

2. Kernel code runs without overhead because Granary relinquishes control whenever an instrumented module executes kernel code. Granary implements a novel technique for re-gaining control when kernel code attempts to execute module code. Each time the instrumented module code invokes a kernel function, all arguments of the function are *wrapped*. Argument wrappers are type- and function-specific, and ensure that potential module entry points (e.g. module function pointers) are replaced with behaviorally-equivalent values that first yield control to Granary. The type- and function-specific wrappers provide the behavioral watchpoint framework the ability to add type information with the watchpoints. They also help the watchpoint framework identify kernel allocators and wrap them to add watchpoints on newly allocated objects.

3. Granary does not require any changes in the module or the kernel code. Granary's wrapping mechanism is portable across different kernel versions because the majority of wrappers are automatically generated by parsing kernel headers and using several meta-programs.

Granary interposes on the Linux kernel's module loading process. When a kernel module is loaded, Granary bootstraps by translating the first basic block of the module's initialisation function. It then replaces the pointer to that function with a pointer to the translated basic block. The translation process continues when the kernel initialises the module by invoking the translated module code.

Granary translates and instruments module binaries on demand (one basic block at a time). The new basic blocks are decoded and translated as execution "discovers" those basic blocks. Translated basic blocks are linked together and stored in a globally accessible *code cache*. Granary's

just-in-time translation approach means that code executing from the code cache may yield control to Granary to request the address of the next basic block to execute. When instrumented code yields to Granary, a "context switch" occurs that transfers execution to a CPU-private stack where Granary operates. Granary context-switches back to the code cache when the next basic block has been found or translated so that instrumented execution may continue.

Granary provides three important features that are helpful in developing DBT-based program analysis tools.

2.1 Mixed-Mode Execution

Granary supports two modes of execution: instrumented and native. Module code is instrumented and executes from Granary's code cache, which is under Granary's control. Non-module kernel code runs natively. A mode switch occurs when execution transfers between native and instrumented code. Some mode switches happen naturally (e.g., when instrumented code returns to native code) and other mode switches are mediated by Granary (e.g., when instrumented module code invokes a kernel function).

The mode switch from instrumented module code to native kernel code is easier to detect since the instrumented code runs under Granary's control. Granary treats all kernel functions as *detach* points, where a mode switch from instrumented to native code occurs. At these points, Granary stops executing. However, Granary needs a way to regain control when native kernel code invokes module code. Granary uses static analysis information to dynamically discover attach points by wrapping the kernel/module interface. Kernel functions are wrapped by *kernel wrappers* that inspect and traverse argument pointers in search of pointers to module functions and replaces them with a function-specific module wrapper. Figure 2.1 shows an example of a type wrapper for Linux device driver structure which is used for wrapping function pointers. After the module initialisation process, the *attaching* of Granary when module code starts executing, happens in one of three ways:

```
struct device_driver {
  ...
  int (*probe)(struct device *);
  int (*remove)(struct device *);
  void (*shutdown)(struct device *);
  int (*suspend)(struct device *,
            pm_message_t);
  int (*resume)(struct device *);
  ...
  const struct dev_pm_ops *pm;
  ...
};
```

```
TYPE_WRAPPER(struct device_driver, {
    PRE_OUT {
        WRAP_FUNCTION(arg.probe);
        WRAP_FUNCTION(arg.remove);
        WRAP_FUNCTION(arg.shutdown);
        ...
    }
    POST_OUT {
        POST_WRAP(arg.pm);
    }
})
```

Figure 2.1: The Linux device driver structure is shown on the left. The automatically generated type wrapper for this structure is shown on the right. In the wrapper code, `arg` is a reference to a `struct device_driver` object passed as, or referenced by, an argument to a kernel or module wrapper. Code in the `PRE_OUT` section is applied to arguments of the wrapped type before a kernel wrapper is invoked. Similarly, code in the `POST_OUT` section is applied to arguments of the wrapped type after a kernel wrapper is invoked. `POST_WRAP` invokes the type wrapper that is specific to the value to which it is applied (`arg.pm`). Type wrappers also support `_IN` suffixes instead of `_OUT` suffixes, which apply to data going into modules (i.e., over module wrappers). Finally, the `RETURN_` prefix is used to apply some code to return values of either kernel or module wrappers.

1. **Implicit attaching:** It happens when kernel returns to the module. The native kernel code returns to instrumented module code in the code cache. This is done at the cost of return address transparency, i.e., code cache addresses are visible on returns to the executing module code.

2. **Fast attaching:** When the kernel invokes a wrapped module function. The addresses of

these module wrapper functions are provided to the kernel through type wrapping at the interface.

3. **Slow attaching:** When the kernel invokes unwrapped module code. If the kernel executes an uninstrumented module function that was passed to the kernel in a type-unsafe manner, then the processor will raise a fault because Granary uses hardware page protection to prevent module code from being executed. Granary handles these faults by returning execution to the instrumented version of the faulting module code.

Granary *detaches* when control transfers from instrumented code to native (uninstrumented) code. Detaching occurs in one of two ways:

1. **Implicit detaching:** Instrumented module code returns to the original kernel, or is interrupted (initial interrupt handling is done by the kernel and hence interrupt handling runs uninstrumented code).

2. **Wrapped detaching:** Instrumented module code invokes a kernel wrapper that later transfers control to the kernel.

Mixed-mode execution provides Granary the ability to instrument only module code. However the attach/detach mechanism of Granary is more generic and it allows the instrumentation tool to switch the execution mode anytime. This selective instrumentation feature is used by the behavioral watchpoint framework to implement on-demand instrumentation. Granary's wrapper provides instrumentation tools information that is derived from static analysis of the kernel.

2.2 Policy-Driven Instrumentation

Granary provides the application the ability to specialise the instrumentation based on execution context. Granary tracks the execution context of a program using technique called *policy-driven*

instrumentation. It allows applications to manage different instrumentation policies which can be dynamically switched as and when required based on the context. The behavioral watchpoint framework uses *policy-driven instrumentation* to track the execution context of the module to add watchpoints on the selected objects. The policy-driven instrumentation in the leak detector helps the watchpoint framework track the module entry and exit path and identify the context of memory allocation. The framework adds watchpoint with newly allocated object only if it happens in the module context.

Granary implements policy tracking and switching by encoding policy information in the meta-data and control transfer instructions (CTIs) of basic blocks. The policy information propagates through the control transfer instruction and the targeted basic block either inherits or switches to the new policy as specified.

2.3 Reifying Instrumentation

Granary uses a technique called *reifying instrumentation* to provide the benefits of high-level static analysis information to dynamic instrumentation tools. Reifying instrumentation bootstraps on Granary's mixed-mode execution approach, which exposes static type information to instrumentation tools. The type information can be used by the behavioral watchpoint framework to identify and add context specific information with the watched objects. The behavioral watchpoint framework actively uses the context-specific information associated with the watched objects to create various analysis tools such as selective memory shadowing for generating the models of typical module behavior.

The high-level static information present at the wrapper serves two roles in the tool. First, the type wrappers are used to assign type-specific IDs to module-allocated memory that is shared across the module/kernel interface. This type-specific ID assignment is critical because it allows us to match memory reads and writes to specific kernel data structure fields. Second, the kernel and module function wrappers are used to generate call graphs of a module's execution. This call

graph models module behaviour (according to the kernel) because we label module code nodes with kernel data structure and function pointer field names (derived from module wrappers). This labelling allows us to generalise across similar modules. For example, code reached by calling the `ext4_mount` and `btrfs_mount` functions from the `ext4` and `btrfs` modules, respectively, are both labelled as `file_system_type::mount`, because these function addresses are stored (and later replaced by module wrappers) in the `mount` field of a `file_system_type` data structure.

These three features make Granary unique among DBT-systems. We choose Granary as the underlying DBT system, this is because of its ability to integrate the high-level static analysis informations with low-level instruction manipulation and support the context-aware runtime code specialisation. These are important to implement different features of behavioral watchpoints. In next chapter, we will discusses the design, implementation and the different features enabled by the behavioral watchpoint that helps in developing program analysis tools.

Chapter 3

Behavioural Watchpoints

Behavioral watchpoint provides an efficient software-based watchpoint framework that simplifies the implementation of DBT-based program analysis and debugging tools. This chapter describes the design of behavioral watchpoints and the various features enabled by our design. We present the challenges raised by our approach and then describe our implementation of behavioral watchpoints. Two characteristics that defines the watchpoints are:

i) Context-specific information is embedded in each watchpoint. This information is directly available when a watched address is accessed, providing significant versatility in monitoring a large number of memory addresses.

ii) A behavioral watchpoint watches a *range* of addresses, enabling object-granularity watchpoints, i.e., one watchpoint can watch an entire object or memory block.

3.1 Design

The design of the behavioral watchpoint framework is motivated by our aim to provide context-specific information on memory accesses, which helps provide significant versatility when monitoring these accesses. This context specific information is stored in an in-memory data structure and accessed before memory read/write operations. Our design is based on the key observation that the pointers in 64 bit architectures have spare bits. Both AMD64 and Intel x86-64 processors use a 48 bit implementation leaving 16 spare bits that can be used to store pointer metadata information.

We implement watchpoints by adding an extra level of indirection to memory addresses. An unwatched address is converted into a watched address by changing its spare high-order bits. The high order bits of an unwatched address have the value `0xffff`. We call this value the canonical value. For watched addresses, the high-order bits are converted to a non-canonical value that helps 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 (*vtable*) 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.

The design of behavioral watchpoints is shown in Figure 3.1. Our design 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*. The *watchpoint descriptor table* stores a pointer to the watchpoint's descriptor. The key advantage of our design scheme is i) the ability to map watched addresses to unwatched addresses using a simple bitmask and ii) the ability to easily access a watchpoint's descriptor when a watched address is accessed. The high-order 15 bits counter index allows us to use 32K watchpoints at a time and the additional 8 bits of inherited index extends the number of possible watchpoints to 8M. The inherited index is left unchanged when converting an unwatched address into watched.

However, one drawback of our design is that an offset of a watched address can cause the low-order bits to overflow into the inherited index and this will lead a watched address to point

to an incorrect descriptor. One approach to deal with this issue is to assign multiple descriptors for the watched objects holding the same meta-information and putting them in adjacent indices or duplicating the same descriptor entry across adjacent indices. This is possible because inherited indexes make the descriptor table more sparse allowing them to have duplicate entries.

Our design of behavioral watchpoints is geared to the 64 bit x86 (x86-64) architecture. In kernel space on the x86-64 architecture, the addressable memory includes the canonical form of addresses with their 16 high-order bits set to 1. Tagging these higher order bits with the descriptor information converts them to a non-canonical address that triggers a hardware exception when dereferenced. The watchpoint framework takes advantage of the exception to implement behavioral watchpoints.

Our 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). The approach is called code-centric instrumentation because the decision about the code translation and instrumentation is taken based on the execution path. 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 is triggered. We call this approach data-centric instrumentation because code translation and instrumentation is based on watchpoint accesses.

The design of behavioral watchpoints provides the following benefits:

i) ***Multiple watchpoints can watch the same range of memory:*** Two copies of an address (e.g., two pointers to the same object) can be watched separately, so long as the high-order bits index different entries in the watchpoint descriptor table. This is useful for distinguishing between logically different objects that occupy the same memory region. For example, this enables efficient detection of use-after-free bugs without preventing deallocated memory from

being immediately reallocated for another use. This efficiency comes from our ability to have one watchpoint for the freed object and another watchpoint for the newly allocated memory occupying the same space.

ii) ***Watchpoint descriptors are context specific:*** 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 3.1. 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 initialised 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.

iii) ***Watchpoints are viral:*** 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.

3.2 Implementation

Behavioral watchpoints are implemented using Granary, a dynamic binary translation (DBT) framework [16] described in Chapter 2. We implement behavioral watchpoints by adding watchpoint instrumentation before every memory read (load) and write (store) instruction in the code cache. The watchpoint instrumentation actively looks for non-canonical addresses being used as a source or destination address and triggers watchpoint handling code if it encounters them. The watchpoint

framework provides an interface for the client to write the watchpoint handling code efficiently. Our approach of implementing the watchpoint framework has three advantages:

i) The framework makes it easy to add watchpoints with the object. It provides an indirection to the memory address of the unwatched object and tags it with the counter index. It separates the allocation and management of descriptors and provides the client an interface to manage its own descriptors. The framework only manages the global descriptor table and assigns the descriptors to the corresponding indices.

ii) The direct mapping of watchpoints with the descriptors make the descriptor information available when the watched objects are accessed. This is helpful in specialising the instrumentation based on the descriptor's information.

iii) The runtime overhead of watchpoint implementation comes from the watchpoint instrumentation. Our implementation of watchpoint instrumentation has a fixed cost and the overhead does not increase drastically with the increase in the number of watchpoints.

3.2.1 Code-Centric Instrumentation

The Code-centric instrumentation dynamically adds watchpoint instrumentation at every memory reference before putting each basic block in code-cache. Code-centric watchpoint instrumentation detects the dereference of watched addresses and resolves them to their unwatched counterparts before performing the memory operation, thus avoiding any hardware traps that would be generated if a watched address is accessed directly. Code-centric instrumentation has low overhead when compared with trap based instrumentation when watchpoints are expected to be triggered frequently.

We implemented behavioral watchpoints using Granary, which comprehensively instruments all module code. Granary detaches itself at the kernel interface thus providing support for code-centric instrumentation only for module code. A module analysis tool developed using the watchpoint framework may need to track the behavior of objects that are shared between the kernel and a module using watchpoints. For these shared objects, we use code-centric instrumentation for the module code, and trap-based or data centric instrumentation (with the transitive policy, as described in the next section) for the kernel code.

3.2.2 Data-Centric Instrumentation

Data-centric instrumentation is triggered by hardware traps when a watched address is dereferenced. At this point, Granary is attached so that code can be instrumented. We used three policies for detaching instrumentation: i) basic-block instrumentation, ii) function-only instrumentation, and iii) transitive instrumentation.

The basic block instrumentation policy detaches instrumentation at the end of the current basic block. This policy translates the minimum amount of code but causes the highest number of traps.

The function-only policy retains the same instrumentation policy across the body of a function and does not allow the transfer of policy across control-transfer instructions. The framework stops instrumenting and detaches itself at the end of function body (i.e, with the `ret` instruction). Any `call` to other functions detaches the framework temporarily and then the framework is attached once that function returns. The policy instruments a moderate amount of code with a corresponding decrease in the number of hardware traps, compared to the basic block policy.

The transitive policy allows the framework to follow the code-centric instrumentation transferring its instrumentation policy across the `call` or `jmp` instructions; the framework detaches itself when the function returns. The transitive policy instruments code aggressively based on the assumption that once the watched address is dereferenced in an execution path, there is high probability of encountering the watchpoints again in the same function or the functions called by this function. Our evaluation result supports this assumption.

The ability to add selective instrumentation by attaching and detaching the watchpoint framework on demand introduces overhead only when instrumentation is needed. The data-centric approach benefits from the lower overhead of infrequent traps when watched addresses are less likely to be dereferenced, e.g., when watchpoints are added selectively for a few objects.

3.2.3 Instrumentation Optimizations

The implementation of behavioral watchpoints essentially requires monitoring of all memory read (load) and write (store) operations. A basic monitoring framework using dynamic binary instrumentation inserts new instructions before every memory references looking for the watchpoint addresses. The framework also inserts watchpoint instrumentation before every memory operation. A naive implementation of watchpoint instrumentation performs the following operations:

i) Save and restore the registers, including flags registers, that are used or affected by watchpoint instrumentation. We use the stack to spill these registers.

ii) Calculate the referenced address and check if this is one of the non-canonical addresses. This is done by inspecting high-order 16 bits of the addresses. Inject a callback function, if the address is in the non-canonical form. This requires the masking of both user and kernel space addresses. The 47th and 48th bit of the address is reserved and used for this purpose.

iii) Emulate the actual instruction with corrected address to perform the normal load/store operation and continue execution after restoring registers and flags. The instruction emulation is done by replacing the original instruction or by creating a new instruction and providing an indirection.

iv) Continue the normal execution after restoring the registers and flags, if the address is not one of the watched addresses.

The naive implementation of watchpoint instrumentation suffers from high runtime overhead. Granary allows implementing several optimizations for reducing runtime overhead. The watchpoint instrumentation uses the following optimizations:

i) ***Dead Registers Analysis****:* The instrumentation system needs scratch registers for creating and injecting new instructions. These registers can be obtained by spilling them to the stack or to memory locations. The frequent spilling of these registers on stack or memory location increases the cost of instrumentation significantly [36, 27]. Granary provides support for managing live registers. It goes over the basic block, performs the register liveness analysis and collects all the registers that can be safely used without spilling them on the stack. The register manager traverses all the instructions in a basic block moving upward and collecting the source and destination registers making all the non-source and non-base-displacement registers (whole destination registers) as dead.

ii) ***Flag Liveness Analysis****:* In an instrumentation system, the newly injected instructions should not affect the state of the processor. The watchpoint instrumentation ensures this by saving and restoring the status registers before and after the injected instructions. This is costly and so the framework reduces this cost by performing eflag liveness analysis [25]. The watchpoint instrumentation saves and restores the flags register only if it is alive.

iii) ***Eliminate Stack Operations****:* In the x86 architecture, any operation on the stack or an operation involving the stack pointer is also a memory operation. Watchpoint instrumentation is not required for these memory operations unless it is specified by the client. The watchpoint framework identifies all such instructions and prevents them from getting instrumented. Applications such as the stack-based overflow detector require instrumenting any operation involving stack pointers and they need to specifically add such instrumentation.

Figure 3.2 shows the native and instrumented versions of the instructions performing memory operations. The DBT system provides several other optimizations such as *Group Checks*, which

consolidate the two consecutive memory reference checks into a single check if there are no intervening instructions that affect address generation, and *Merge Checks* which exploit the locality of memory references and merge the instrumentation for instructions accessing different members of the same object in the same basic block. We have not implemented all these optimizations in the watchpoint framework because our purpose is not to be exhaustive but rather to demonstrate that watchpoint instrumentation can perform online monitoring of memory references with reasonable overhead.

3.3 Limitations

The design of behavioral watchpoints introduces non addressable memory in the system. The use of non-canonical addresses puts the following restrictions on the way watchpoints can be used. Some of these limitations are application specific and depend on how the program analysis tool uses them.

i) The behavioral watchpoint framework suggests adding watchpoints early in the life of the objects. Adding watchpoints at an arbitrary location should be avoided as it can introduce inconsistency in the program. A program running with both watched and unwatched versions of the same object will provide only partial information about the object. However some program analysis tools such as RCU debugger takes advantage of this limitation and generates two versions of the same objects, each of them getting watched differently.

ii) Behavioural watchpoints uses high-order bits to store the descriptor information. The descriptor provides the *meta-informations* about the object being watched. This puts restriction on the use of behavioral watchpoints in analysing applications which play with the high-order bits of its addresses. Such application will destroy the descriptor information stored with the watchpoints. Applications doing the similar operations such as sign extension of the 64-bit addresses or handling of only 32-bit addresses also loses the descriptor information. The

watchpoint framework has no mechanism to recover the lost descriptor information from the watchpoints. Identifying and ignoring such applications or the operations in an application is an important challenge for the watchpoint framework.

iii) Behavioural watchpoints are implemented for the Linux kernel. The Linux kernel uses bitwise operations for the fast page-table lookup. These bitwise operations converts the virtual page addresses into the page frame number and assumes that the 16 high-order bits of the addresses are all one. Linux kernel also maintains different address space regions and perform different functions for the page table-lookup. The design of behavioral watchpoints are not friendly with these operations.

3.4 Evaluation

In this section, we evaluate the performance overhead of behavioral watchpoints. The watchpoint framework implements behavioral watchpoints by injecting instrumentation for every memory read (load) and write (store) operations, introducing runtime overhead in the system. We evaluated the cost of watchpoint instrumentation, including the cost of handling hardware traps for data-driven instrumentation, using synthetic microbenchmarks.

We also measured the overhead of using behavioral watchpoints on the throughput of common file I/O operations using the *iozone* [32] file system benchmark and the performance of real-world workloads using several file system utilities and the Filebench benchmarks. We ran all our experiments on a desktop equipped with an Intel® Core™ 2 Duo 2.93 GHz CPU with 4GB physical memory. In our experimental setup, we used the `ext3` file system module that was mounted on a 1GB RAMDisk (mounting loads the `ext3` and `jbd` journaling kernel modules). The watchpoint framework wraps the kernel memory allocators to add watchpoints for all newly allocated objects.

Optimizations	Native	Watch Null	Watch All
Code-driven (with BB optimizations)	1x	2.7x	3.8x
Code-driven (without BB optimizations)	1x	5.0x	6.2x
Data-driven approach	1x	1x	283x

Table 3.1: The performance overhead of watchpoint instrumentation on a synthetic microbenchmark (Figure 3.3). The code-driven approach evaluates the cost of using watchpoint instrumentation and the benefit of basic block optimizations. The overhead of the data-driven approach is dominated by the cost of handling hardware traps.

3.4.1 Microbenchmark

Our microbenchmark consists of a tight-loop of memory operations that exhibits the worst-case overhead of using watchpoint instrumentation. Figure 3.3 shows the synthetic microbenchmark we used to evaluate the overhead of watchpoint instrumentation and hardware traps. We also evaluated the benefits of our optimization schemes such as register-liveness analysis and flag-liveness analysis used for watchpoint instrumentation. We call this basic block optimizations.

Table 3.1 shows the maximum cost of using watchpoint instrumentation and handling a hardware trap. *Watch None* represents the overhead of baseline watchpoint instrumentation when no watchpoints have been added, and *Watch All* shows the overhead when all module allocated objects are being watched. The added overhead of *Watch All* comes due to the indirection required for masking watched addresses and emulating the original instructions.

Table 3.1 also shows the high overhead of handling hardware traps. In the data-centric approach, the hardware trap triggers the instrumentation of the module code. The high overhead of a hardware trap shows that the approach is useful for selective instrumentation when watchpoints are less likely to be triggered.

Watchpoint statistics for the code-centric instrumentations		
	Watch None	Watch All
Number of basic blocks	2249	5495
Number of basic blocks with watched memory operations	0	1208
Number of executed basic blocks	238782485	825836778
Number of dynamic memory operations	568643857	1693113417
Number of watched memory operations	0	199495293
Number of kernel hardware traps	0	12630779

Table 3.2: The statistics of the watchpoint framework for code-driven instrumentation when module allocated objects are watched. It also shows the effect of watched objects leaked to the kernel. The increase in the number of hardware traps causes an increase in the number of basic blocks and the number of executed basic blocks.

3.4.2 Iozone filesystem benchmark

We used the *iozone* [32] file system benchmark to measure the overhead of behavioral watchpoints on the throughput of common file system operations. In our experimental setup, we used the `ext3` file system module that was mounted on a RAMDisk of size 1GB. We enabled direct IO to avoid the effect of buffer cache on file system. The *iozone*, in throughput mode, created two processes (reader, writer) that perform the file I/O operations on a file of size 480Mb with a record size of 4Kb. The watchpoint framework wraps the two most commonly used memory allocators in `ext3` and `jbd`: `__kmalloc` and `kmem_cache_alloc`, to add watchpoints on the allocated objects. We evaluated both the code-centric & data-centric approaches using *iozone* and compared their performance.

Code-centric instrumentation: The code-centric approach comprehensively instruments the module code and dynamically adds watchpoint instrumentation at every memory reference. This baseline watchpoint instrumentation causes overhead even when there are no added watchpoints. In the code-centric approach the module always runs under the control of Granary and executes code from the code-cache. It uses the kernel and module function wrappers for fast attaching and detaching at the interface.

In the code-centric instrumentation, we first evaluated the overhead of watchpoint instrumentation and the cost of using Granary as the underlying DBT system. This is important to understand the baseline cost of using the watchpoint framework.

Figure 3.4 represents the overhead of code-centric instrumentation on the throughput of file I/O operations. The overhead of the watchpoint framework increases to ~70% when all objects allocated by the module are watched. This is because many of these watched objects are shared and leak to the kernel. When these objects are dereferenced in the kernel, they cause hardware traps because Granary detaches itself at the kernel interface and the watchpoint instrumentation is no longer added to the memory references. These hardware traps are costly and reattach the watchpoint framework which then instruments the kernel code transitively.

Table 3.2 represents the number of hardware traps encountered and number of executed basic blocks for the code centric instrumentation. The additional increase in the number of basic blocks (~2.5×) and the number of executed basic blocks (~3.5×) comes due to the kernel code instrumentation on hardware traps. It also shows that the one out of every sixteen watched memory operations causes hardware traps.

For understanding the source of these hardware traps, we developed a tool using the watchpoint framework which tracks the accesses of module allocated objects. We typed these objects based on their size and the memory allocator used for its allocation. Table 3.3 shows the access pattern of the module allocated objects. The post processing analysis shows that file system `inode` objects with its size "768" gets accessed by the kernel maximum number of times and adding watchpoint on `inode` objects cause maximum number of hardware traps.

We verified this by removing watchpoints from all `inode` objects and running *iozone* in

Access pattern of module allocated objects			
Memory Allocator	Size	Module Accesses Count	Kernel Accesses Count
$\_\_kmalloc$	8	1233865	0
	50	720	288
	51	648	240
	59	792	528
	4096	23452	0
$kmem\_cache\_alloc$	16	31533	0
	24	25019188	0
	32	2348	0
	64	2469028	3099
	112	21404553	0
	192	19902520	4916028
	768	57760206	37373718
	1024	34865132	7940485
	8196	299	821648
$\_get\_free\_page$	4096	820273	2973

Table 3.3: The memory accesses patten of the module allocated objects. It represents the number of times an object is getting accessed by the module and the kernel code. The objects are classified based on its size and the memory allocator used for allocation.

Watchpoint statistics for the code-centric instrumentations		
	Watch none `inode`	Watch All
Number of basic blocks	2874	5495
Number of basic blocks with watched memory operations	732	1208
Number of executed basic blocks	266083908	825836778
Number of dynamic memory operations	594195510	1693113417
Number of watched memory operations	92600064	199495293
Number of kernel hardware traps	5285701	12630779

Table 3.4: The watchpoint statistics of the framework for code-driven instrumentation when all objects allocated by the modules (except `inodes`) are watched.

throughput mode. Figure 3.5 shows that the overhead for *Watch All* decreases by ~50% and is close to the overhead of baseline instrumentation i.e, *Watch None*. The small differences in the overhead of *Watch All* and *Watch None* are because there were still some shared objects which was getting watched and causing hardware traps. Table 3.4 shows the details about such objects.

The evaluation of code-centric instrumentation shows that the hardware traps on the kernel code are the major source of overhead in implementing behavioral watchpoints. These hardware traps can be removed by providing full kernel support to the code-centric instrumentation. Granary, being the underlying DBT-system, currently does not provide support for whole kernel instrumentation.

Data-centric instrumentation: We implemented data-centric instrumentation with three detach policies: i) transitive policy, ii) function-only policy and iii) basic block policy. Section 3.2.2 discusses each of the three policies in detail.

Watchpoint statistics for data-centric instrumentation (with different detach policies)			
	transitive	function	block
Number of basic blocks	5755	3335	1144
Number of basic blocks with watched memory operations	1479	1218	1141
Number of executed basic blocks	833167407	108965164	37378155
Number of dynamic memory operations	1702295819	565666812	251985370
Number of watched memory operators	336149753	234340514	206334774
Number of kernel hardware traps	13403663	42259345	97518601
Number of module hardware traps	112172	23496407	42902684

Table 3.5: The statistics of the watchpoint framework for data-centric instrumentation when all the objects allocated by the modules are watched. It compares the three detach policies used by the data-centric approach.

We evaluated the overhead of data-centric instrumentation with *iozone*, using the same experimental setup as with the code-centric instrumentations. We disabled the kernel and the module wrappers since they allow Granary to take complete control over the module code and prohibit the module code from running native. In data-centric instrumentation, the module code gets executed natively when there are no added watchpoints.

Before comparing the code-centric and data-centric approaches, we first evaluated the three detach policies of data-centric instrumentation. Figure 3.6 shows the overhead of each policy on the throughput of file I/O operations. The transitive policy instruments code aggressively and performs better than the function only and basic block detach policies. Table 3.5 shows the number of executed basic blocks and the number of hardware traps encountered when each of the three detach policies are used. The transitive policy encounters the minimum number of hardware traps and executes the largest number of basic blocks.

Figure 3.7 compares the overhead of code centric and data centric instrumentation. It shows that the code centric instrumentation performs better than the data centric instrumentation when all objects allocated by the module are watched. This is because the code centric instrumentation handles less hardware traps, which is costly and affects the performance of the system. Our evaluation compares the code centric instrumentation with the data centric instrumentation using transitive detach policy.

Selective instrumentation: The data centric approach enables selective instrumentation by adding watchpoints on selected objects. We first evaluated the performance of selective instrumentation by adding watchpoints only on file `inode` objects. Figure 3.8 shows the overhead of selective instrumentation and compares it with the overhead of data-centric instrumentation when all the module allocated objects are watched. Selective instrumentation performs better since it adds less watchpoints thus encountering fewer hardware traps, and executing fewer instrumented basic blocks.

We also compared the performance of selective instrumentation when using code-centric and

data-centric approaches. Figure 3.9 compares the overhead of selective instrumentation for both the approaches. It shows that selective instrumentation using data centric instrumentation has less overhead than with code centric instrumentation. This is because fewer basic blocks gets executed with data-centric instrumentation. The number of hardware traps encountered in both the approaches are also approximately same, causing similar overhead.

The evaluation of code-centric approach shows that a major source of overhead in code-centric instrumentation comes due to adding watchpoints on file `inode` objects. These objects gets accessed mostly by the kernel code causing hardware traps. We removed the effect of these hardware traps from code-centric instrumentation by adding watchpoints selectively on the objects accessed only by the modules. We took the help of table 3.3 for adding watchpoints on the objects selectively.

Figure 3.10 represents the overhead of selective instrumentations for data-centric approach and compares it with the code-centric instrumentation. It shows that the selective instrumentation reduces the overhead of using the watchpoint framework to ~5% where as the baseline instrumentation for code-centric approach causes an overhead of ~40%. This is because the selective instrumentation executes less number of basic blocks (~2.3×). The evaluation also shows that the performance of sequential write operation in case of data-centric approach is less than the code-centric. We analyzed the behavioral of write operations and found that ~50% of the total hardware traps was only happening during sequential write operations. This could be a reason of sequential write not performing better.

The evaluation of both, the code-centric and data-centric approaches for implementing behavioral watchpoints, shows an interesting trade-off between the amount of instrumented code executed from the code-cache and how much of that code actually needs to be instrumented. The code-centric approach always suffers from the overhead of baseline instrumentation since it always executes instrumented basic block from code cache. The data-centric instrumentation overcome this by adding instrumentation on-demand. However, it suffers from the high cost of the hardware traps which triggers the instrumentation. The number of executed basic blocks also increases with the hardware traps which causes additional overhead.

We also saw that the number of hardware traps depend on the type of the watched objects. Adding watchpoints on frequently accessed shared objects such as file `inodes`, causes a large number of traps affecting the performance of the watchpoint framework where as watching relatively less accessed objects such as `ext3_block_alloc_info` causes less traps improving the performance of the data-centric instrumentation.

The advantages and disadvantaged of both the approaches opens up the space to further explore the possibility of switching between the two approaches at runtime. This will reduce both the overhead of baseline instrumentation and the cost of hardware traps, thus improving the performance of the system. The evaluation of code-centric instrumentation also shows its need to provide the whole kernel instrumentation support. We plan to take this as an immediate future work.

3.4.3 Macrobenchmark

We further evaluated the overhead of the behavioral watchpoint framework using a file system macrobenchmark consisting common system utilities. We used the same experimental setup and mounted the `ext3` module on a RAMDisk of size 1GB. The macrobenchmark operated on the Linux source tree (`linux-3.2.50`). Table 3.6 shows the overhead of behavioral watchpoints on the standard system utilities. For both the code-centric and data-centric approaches, the framework adds watchpoints selectively on all `inode` objects allocated from the look-aside buffer. The evaluation shows data-centric instrumentation is performing better than code-centric instrumentation when used for watching objects selectively all the `inode` objects.

In both the code-centric and data centric approaches, the major factors which affects the performance are the number of basic block executed and the number of hardware traps encountered during the execution. We also used microbenchmark to study the changes in the number of hardware traps and the basic block executed with the change in the number of watchpoints. Figure 3.11 shows that both the code centric instrumentation and data centric instrumentation encounters same number of hardware traps. This is because we add watchpoints selectively on all `inode` objects which gets accessed both by the module and the kernel code equally causing hardware traps. We

	Native execution	Data centric approach	Code centric approach
cp	6.64	10.82	11.61
tar	2.64	3.84	4.16
stat	3.90	5.12	5.44
grep	3.94	5.36	5.54

Table 3.6: The CPU (system & user) time for the standard system utilities performing file system operations on a RAMDisk of size 1GB. The framework adds watchpoints selectively on all `inode` objects of the `ext3` module.

also see fewer occurrences where the number of traps in case of data centric is less than the code centric instrumentation. One reason for this could be that the code centric instrumentation always detaches itself at the interface where as the data centric instrumentation, with transitive detach policy, does not always detaches itself at the interface.

Figure 3.11 also shows that with fewer watchpoints the code centric instrumentation still executes many more basic blocks than the data centric approach. This causes increased overhead for code centric instrumentation even when fewer watchpoints are added.

3.4.4 Server Benchmark

We used the Linux port of Filebench version 1.4.9, with four of the standard server workload personalities, to evaluate the watchpoint framework. We used the same experimental setup as with the *iozone* and enabled default settings to define the workload characteristics. The relevant characteristics for the workloads are shown in Table 3.7. With the default characteristics, the datasets easily fit in the RAMDisk used for evaluation. This does not limit the workload with the performance of I/O operations.

Workload	Setting	Data Size
Fileserver	nfiles=10K	1.2GB
Webserver	nfiles=1K,	14.76MB
Webproxy	nfiles=10K	154MB
Varmail	nfiles=1K	14.76MB

Table 3.7: Benchmark characteristics.

Figure 3.12 shows the throughput of the different Filebench server workloads. For evaluation we used data-centric instrumentation with all file `inode` objects getting watched. The data centric instrumentation reduces the throughput for all the workload personalities by 40%. We also evaluated the code-centric instrumentation using Filebench and noticed similar drop in the throughput but we are not showing it here.

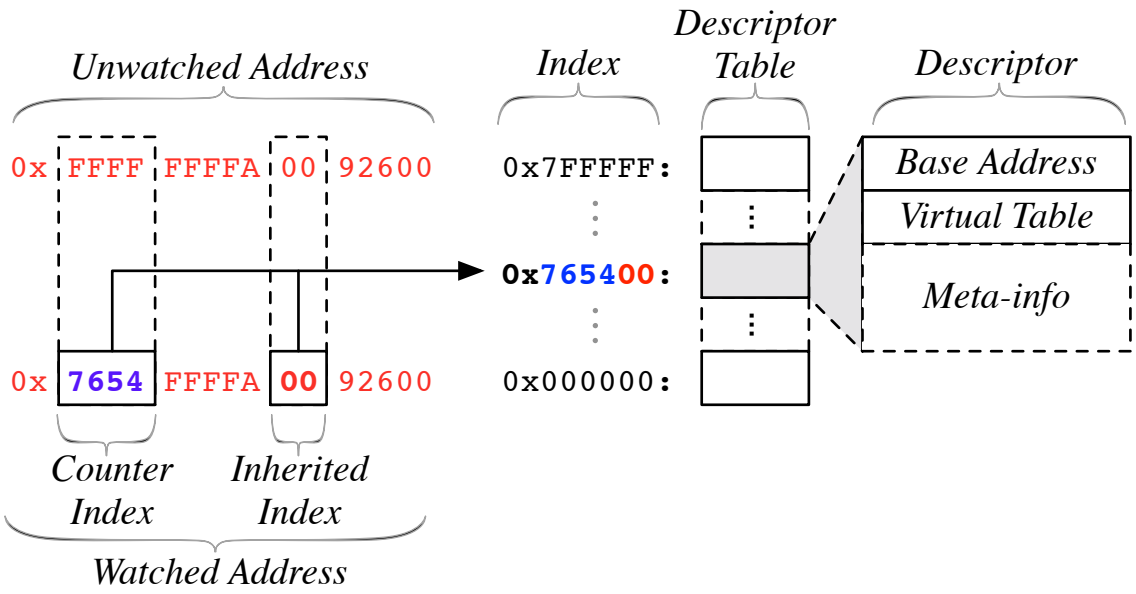


Figure 3.1: A watched address (bottom left) and its corresponding unwatched address (top left) are compared. The watched address takes the form of non-canonical address that are not addressable in kernel space. The watchpoint framework uses a *counter index* and an *inherited index* to store the descriptor information in a global descriptor table. The process of resolving the watchpoint descriptor is shown.

Listing 3.1: Original instructions

```
1  mov    %rbx,0x340(%r13)
2  mov    $0x400,%esi
3  mov    %rax,0x80(%rbx)
```

Listing 3.2: Translated instructions

```
1  lea    0x340(%r13),%rsi
2  bt     $0x30,%rsi
3  jb     addr_not_watched_2
4  bt     $0x2f,%rsi
5  jae    addr_not_watched_2
6  callq  granary_bounds_check_8_rsi
7  bswap  %rsi
8  mov    $0xffff,%si
9  bswap  %rsi
10 LABEL: addr_not_watched_2
11 mov    %rbx,(%rsi)
12 mov    $0x400,%esi
13 push   %rcx
14 lea    0x80(%rbx),%rcx
15 bt     $0x30,%rcx
16 jb     addr_not_watched_3
17 bt     $0x2f,%rcx
18 jae    addr_not_watched_3
19 callq  granary_bounds_check_8_rcx
20 bswap  %rcx
21 mov    $0xffff,%cx
22 bswap  %rcx
23 LABEL: addr_not_watched_3
24 mov    %rax,(%rcx)
25 pop    %rcx
```

Listing 3.3: Overflow detector

```
1  granary_bounds_check_8_rsi :
2  pushfq
3  push   %rax
4  push   %rdi
5  push   %r8
6  mov    %rsi,%rdi
7  mov    %rsi,%r8
8  mov    %rsi,%rax
9  shl    $0x24,%r8
10 shr    $0x38,%r8
11 shr    $0x31,%rdi
12 shl    $0x8,%rdi
13 or     %r8,%rdi
14 lea    0x192c98(%rip),%r8
   # 0xffffffffa03cf280 <client::wp::DESCRIPTORS>
15 lea    (%r8,%rdi,8),%rdi
16 mov    (%rdi),%rdi
17 cmp    (%rdi),%eax
18 jl     overflow_detected
19 mov    %rax,%r8
20 add    $0x7,%r8
21 cmp    %r8d,0x4(%rdi)
22 jle    overflow_detected
23 jmp    no_overflow_detected
24 LABEL : overflow_detected
25 callq  buffer_overflow_handler
26 LABEL : no_overflow_detected
27 pop    %r8
28 pop    %rdi
29 pop    %rax
30 popfq
31 retq
```

Figure 3.2: The native and instrumented version of the instructions performing the basic memory operations. The translated instruction (Listing 3.2) shows the watchpoint instrumentation required for detecting watchpoints and performs the memory operation at the watched addresses. The buffer overflow detector (Listing 3.3) gets triggered only if the addresses are watched. The overflow detector makes a call to `buffer_overflow_handler` if the overflow is detected.

```
typedef unsigned long long cycles_t;
cycles_t currentcycles() {
    unsigned cycles_low, cycles_high;
     __asm__ __volatile__ (
                        "cpuid\n\t"
                        "rdtsc\n\t"
                        "mov␣%%edx,␣%0\n\t"
                        "mov␣%%eax,␣%1\n\t": "=r" (cycles_high),
                        "=r"(cycles_low):: "%rax", "%rbx", "%rcx", "%rdx");

    return ((unsigned long long)cycles_low)
                | (((unsigned long long)cycles_high) << 32);
}
enum {
        NUM_OUTER = 10,
        NUM_INNER = 100000
};

/* Initialize the LKM */
volatile unsigned long x[2] = {100, 200};
int init_module() {
    int i,j;
    unsigned long addr;
    unsigned long flags;
    volatile struct foo_test *ptr =
                (struct foo_test*)kmalloc(sizeof(struct foo_test), GFP_KERNEL);

    cycles_t before[NUM_OUTER], after[NUM_OUTER];

    for(j=0; j < NUM_OUTER ; j++)
    {
        preempt_disable();
        raw_local_irq_save(flags);
        before[j] = currentcycles();
        for(i = 0; i < NUM_INNER; ++i){
                __asm__  volatile("mov␣%rax,␣%rax;");
                (ptr)->l1 += x[i % 2];
                __asm__  volatile("mov␣%rax,␣%rax;");
        }
        after[j] = currentcycles();
        raw_local_irq_restore(flags);
        preempt_enable();
    }

    return 0;
}
```


Figure 3.3: Synthetic microbenmark used to evaluate the watchpoint instrumentation. It performs basic memory operation inside a tight-loop.

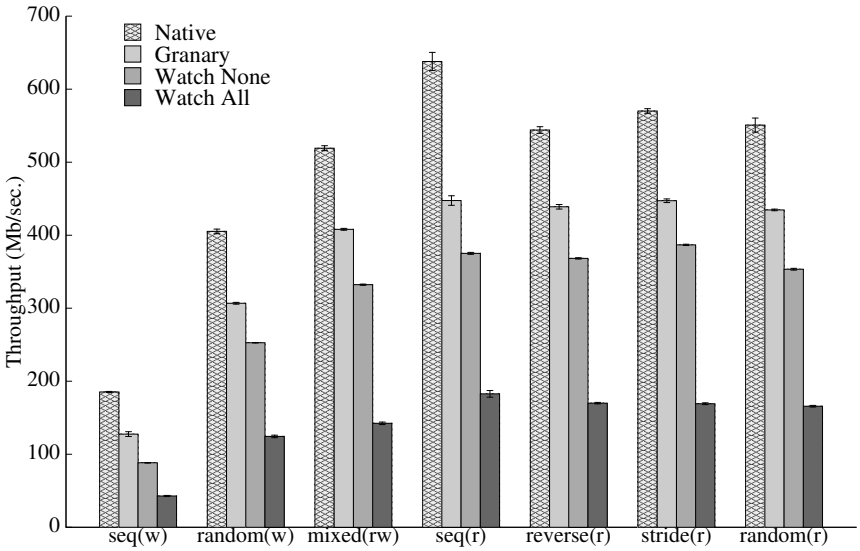


Figure 3.4: Throughput in MB/sec of common file system operations for code-centric instrumentation. Direct IO is enabled to bypass the effect of the OS buffer cache on read requests. It compares the overhead of the watchpoint framework with the performance overhead of Granary and with native system.

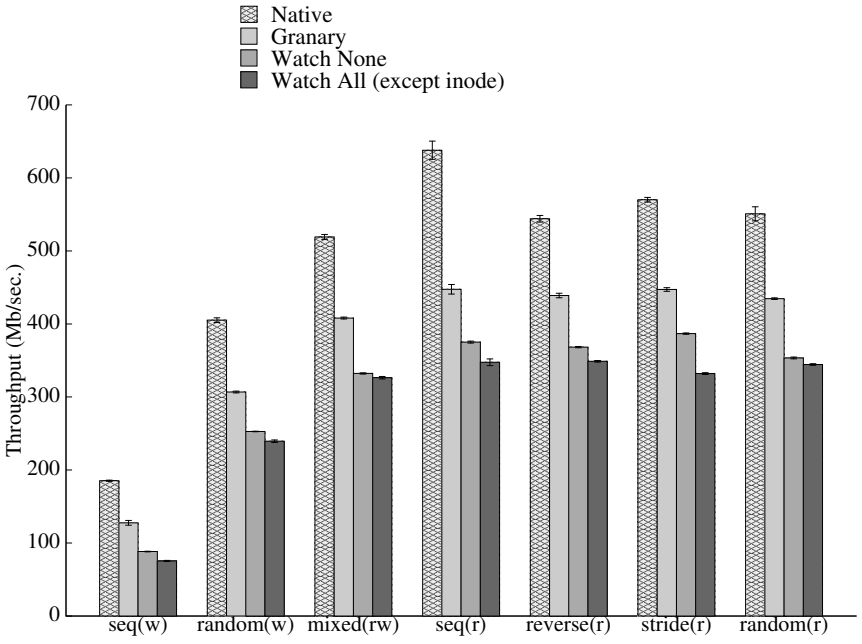


Figure 3.5: Throughput in MB/sec of common file system operations for code-centric instrumentation when `inode` objects are not getting watched. The overhead of *Watch All* decrease because of the less number of hardware traps and executed basic blocks.

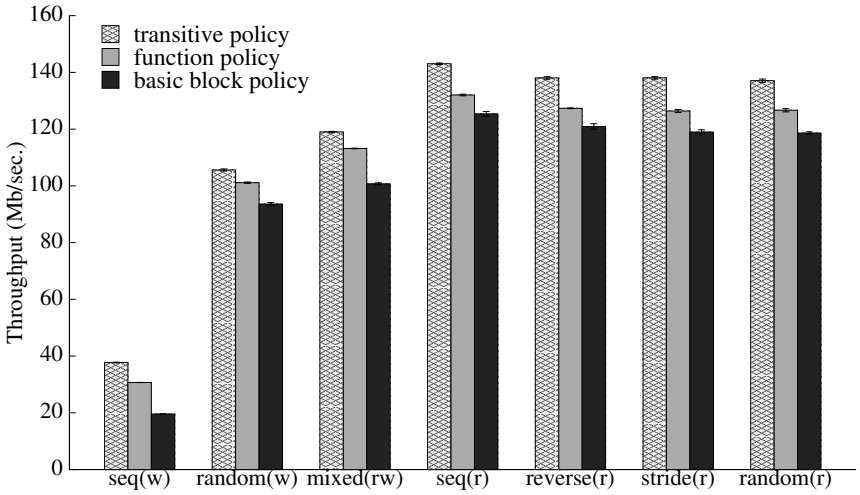


Figure 3.6: Throughput in MB/sec of common file system operations for data-centric instrumentation. It shows the overhead of three data-centric policies: transitive policy, function only, and basic block only policy.

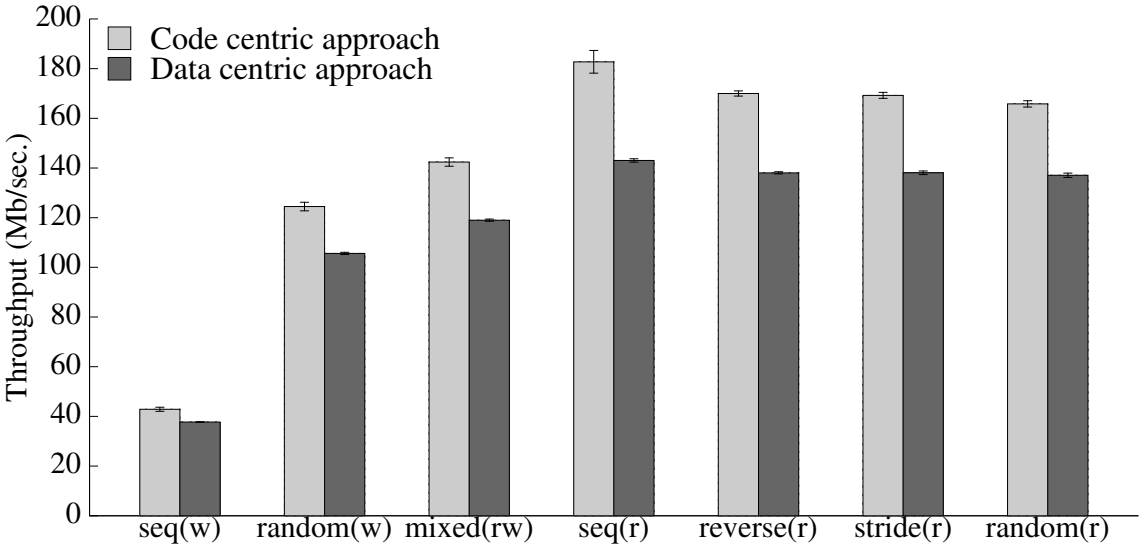
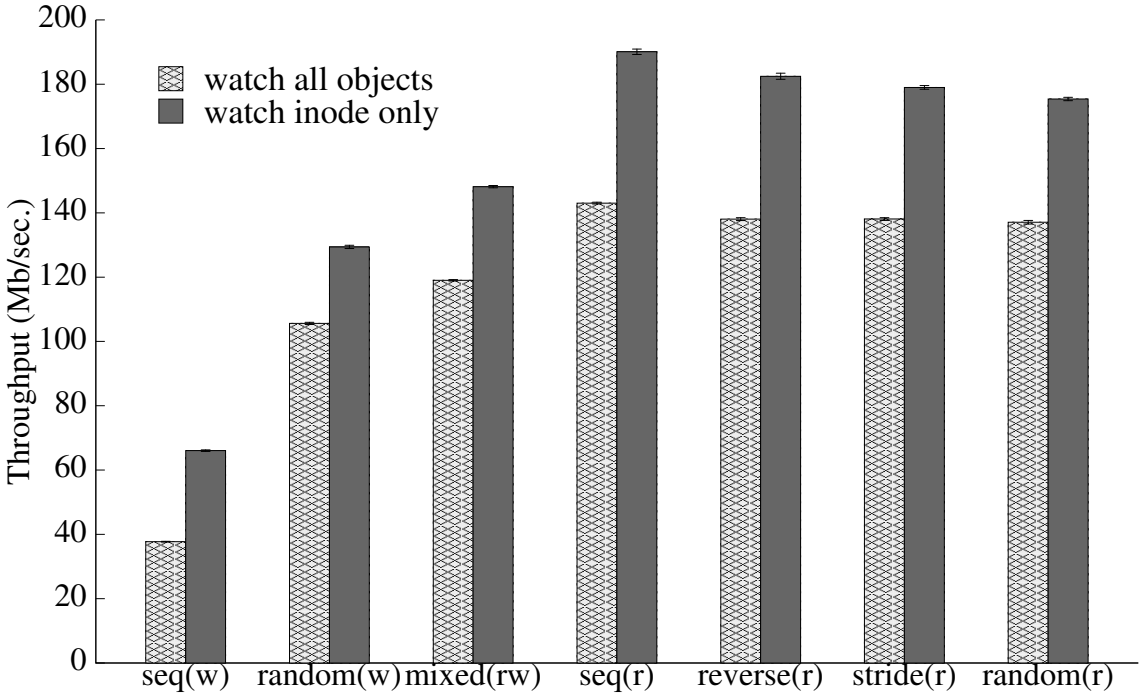
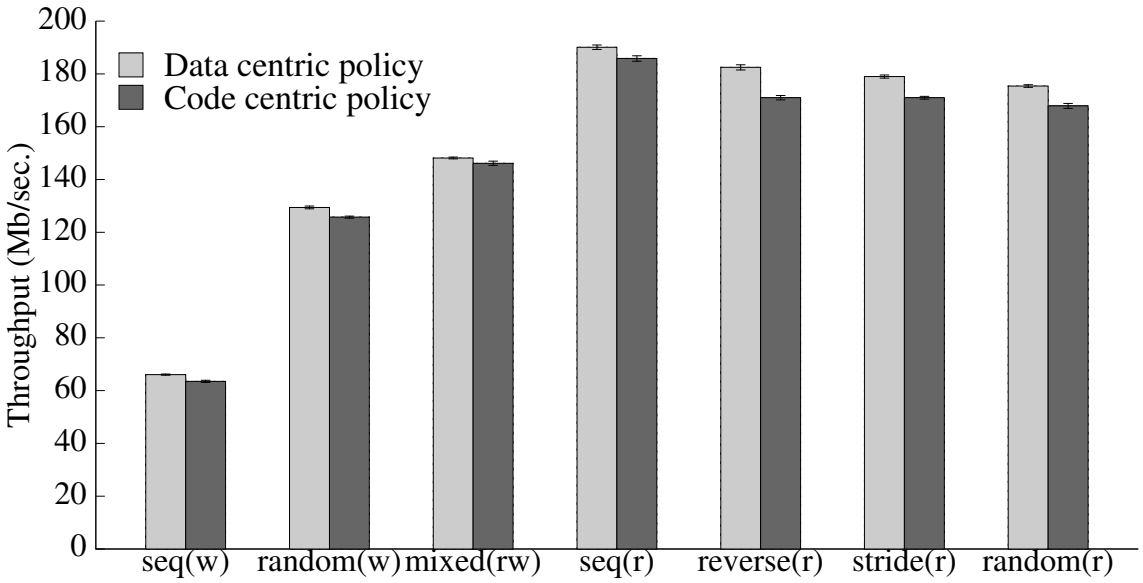


Figure 3.7: Throughput in MB/sec of common file system operations for the code-centric and data-centric approaches. The watchpoints are added on all objects allocated by the module.



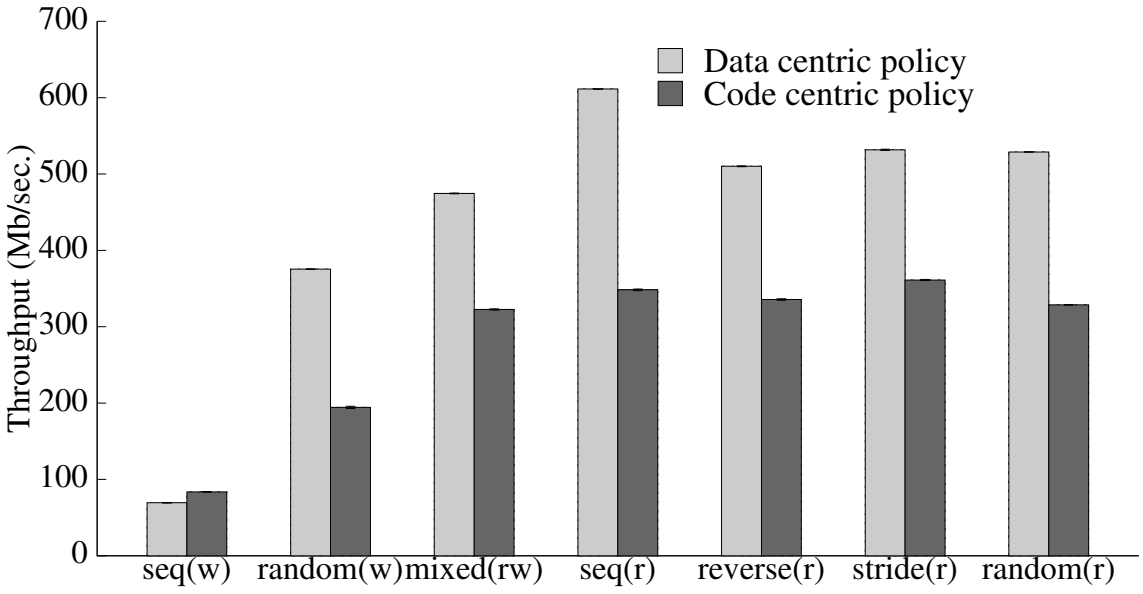
Watchpoint statistics for selective instrumentation in data centric instrumentation		
	Watch `inodes`	Watch all
Number of basic blocks	4217	5755
Number of basic blocks with watched memory operations	562	1479
Number of executed basic blocks	735367411	833167407
Number of dynamic memory operations	1519057148	1702295819
Number of watched memory operators	87433900	336149753
Number of kernel hardware traps	5522314	13403663
Number of module hardware traps	11234	112172

Figure 3.8: Throughput in MB/sec of common file system operations and the watchpoint statistics for data-centric instrumentation when watchpoints are added selectively and only for `inode` objects. The evaluation is done for transitive detach policy.



Watchpoint statistics for selective instrumentation (watch `inode` objects)		
	Code centric	Data centric
Number of basic blocks	5405	4217
Number of executed basic blocks	814496751	735367411
Number of dynamic memory operations	1684738365	1519057148
Number of watched memory operators	91699675	87433900
Number of kernel hardware traps	5446745	5522314
Number of module hardware traps	0	11234

Figure 3.9: Throughput in MB/sec of common file system operations and the watchpoint statistics when watchpoints are added selectively and only on `inode` objects.



Watchpoint statistics for selective instrumentation (no hardware traps on kernel code)		
	Code centric	Data centric
Number of basic blocks	2259	1171
Number of basic blocks with watched memory operations	179	189
Number of executed basic blocks	238740476	105345884
Number of dynamic memory operations	568556578	221904835
Number of watched memory operators	45649624	45637811
Number of module hardware traps	0	10244480

Figure 3.10: Throughput in MB/sec of common file I/O operations and the statistics of the watchpoint framework when watchpoints are added to the objects selectively which are not getting accessed by the kernel. This removes the effect of hardware traps from the code-centric instrumentations.

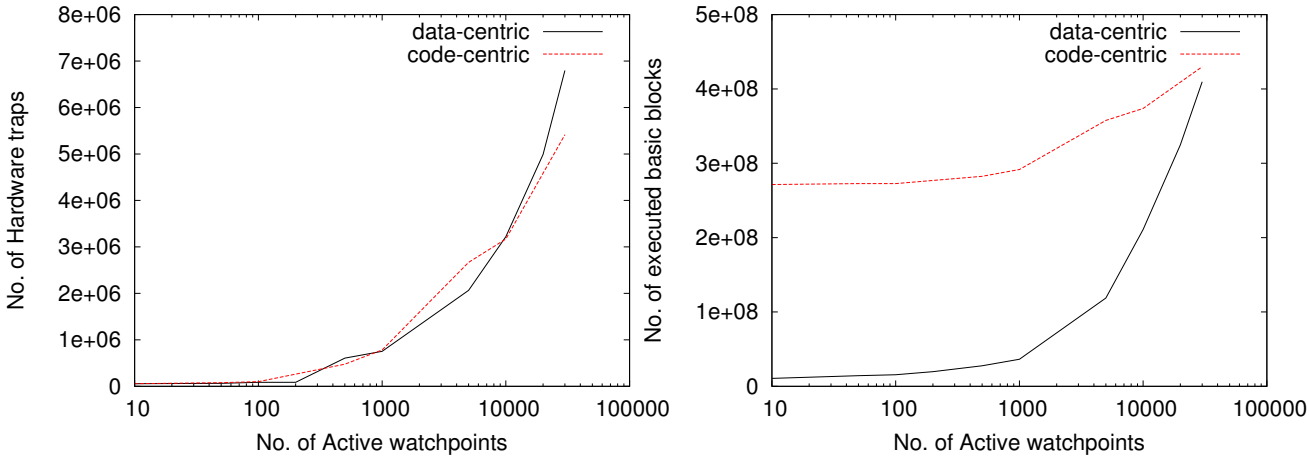


Figure 3.11: The changes in the number of hardware traps and the number of executed basic blocks with the changes in number of added watchpoints.

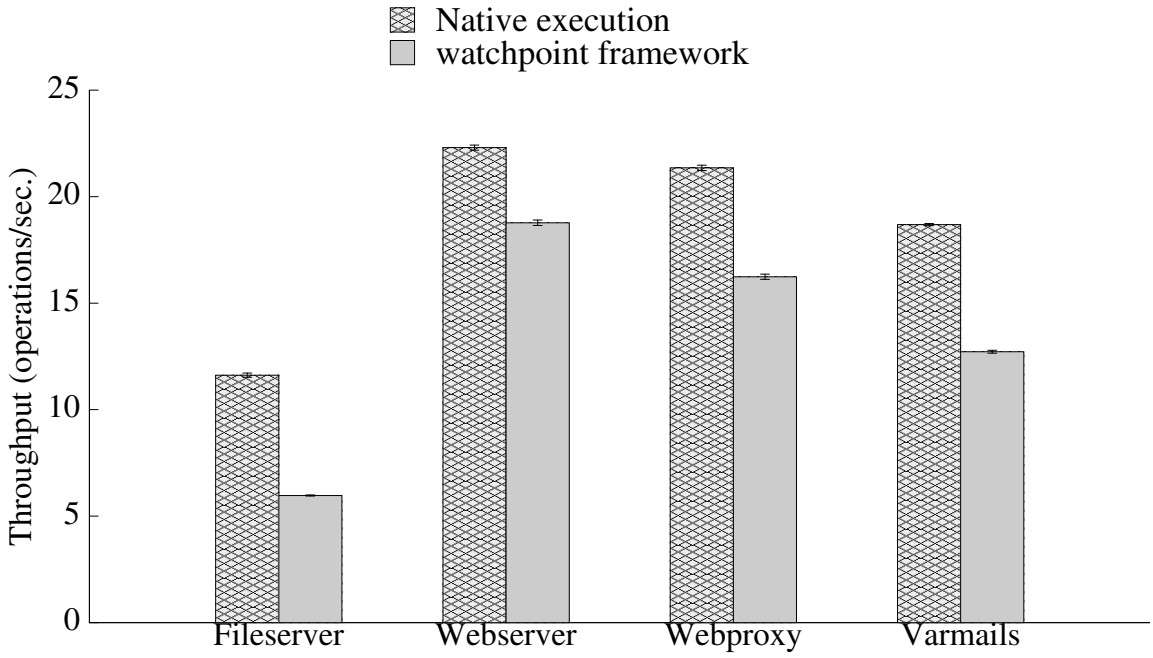


Figure 3.12: Throughput in file system operations/sec for the filebench workload personalities shown in Table 3.7.

Chapter 4

Applications

In this chapter, we describe several prototype applications that we have developed using the behavioral watchpoint framework. These applications provide debugging facilities for kernel modules and demonstrate the effectiveness of the behavioral watchpoint framework. This chapter uses the term *object* to refer to a range of memory locations that are allocated together as a unit.

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. A low-level programming language like C provides raw memory pointers, permits pointer arithmetic, and does not check bounds when accessing arrays. This can result in very efficient code, but the unfortunate side-effect is that accidentally accessing wrong memory is a very common programming error.

The most obvious example in this class of errors is exceeding the bounds of an array. However the bounds of non-array data objects such as heap blocks, C structs, and stack frames, can also be

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

Figure 4.1: 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.

violated. We consider any memory access that falls outside its intended memory range as bound errors. These errors are not difficult to introduce, and they can cause several bugs, some of which can be extremely subtle and lurk undetected for years. Because of this, tools for preventing and identifying them are extremely useful.

Existing solutions such as Memcheck [28] and *fat pointers* [19], provide a powerful facility for detecting bounds errors. *fat pointers* hold meta-data for each pointer with the pointer. This metadata describes the memory ranges the pointer can legitimately access and any access via a pointer that is outside its legitimate range is flagged as an error. Another method of detecting buffer overflows relies on the compiler to allocate "poisoned" regions of memory around each object [40]. Small overflows (e.g., off-by-one errors) are detected by this approach because they access poisoned memory. Large overflows that "skip" over poisoned memory and access nearby memory objects are not detected.

Our implementation of a buffer-overflow detector takes an approach similar to *fat pointers* but we store the meta-information for each object in its watchpoint descriptor. 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, even when objects are adjacent to each other.

We employ three overflow detection policies: heap-based, type-based and stack-based. All three policies depend on the same extension to the meta-information associated with watchpoint descriptors: a *limit address*. Together, the base address (stored in the descriptor) and the limit address delineate an object's boundaries in memory [19].

4.1.1 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 4.1. 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 [19]. A buffer overflow is detected when a dereference of a watched address occurs outside of the bounds recorded by the watchpoint's descriptor (Figure 4.2). 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, as shown in case (3) of Figure 4.2.

We use Granary to substitute invocations of the kernel's memory allocators (e.g. `kmalloc`) with wrapped versions of the allocators (Figure 4.1). A wrapped allocator invokes the original allocator but returns a watched version of the allocated address. The lifetime of the watchpoint is associated with the lifetime of the objects and watchpoint descriptor gets collected when objects get freed. The watched address returned has its descriptor initialised with the base address as the allocated address and with the limit address as the base address plus the requested allocation size. All buffer overflow detecting watchpoints are initialised with the same vtable pointer, which points to a table of memory operand size-specific functions. The operand size is necessary to detect dereferences of memory that overlap both the object and its adjacent memory, as illustrated in case *(3)* of Figure 4.2. The vtable function corresponding to the memory operand size is invoked when a watched address is dereferenced. Each vtable function is programmed to check the watched address against the base and limit addresses stored in watchpoint's descriptor (Figure 4.2). If a

buffer underflow or overflow is detected then the vtable function notifies the run-time system.

4.1.2 Type-based overflow detection

Type-based overflow detection is applied when the type of an object is known. It is distinguished from heap-based overflow detection in that it can apply both to heap and non-heap memory, and it can be used to inform the run-time system about more accurate bounds information. In all other respects, the type-based overflow detection policy detects bugs in the same way as the heap-based policy (as might be encoded within the fields of a vector-like data structure). If the type of an object is known then it can reveal semantic relationships which can enable further type- or heap-based overflow detection. For example, the fields within a structure might encode memory bounds information (as would be the case for a vector-like data structure)

The simplest use case of type information is to infer the limit address of a typed pointer. A more complex use case arises when the fields within a structure encode memory bounds information. In both cases, the vtable functions for typed and untyped memory operate in the same way. The type of an addressable object becomes known to Granary in the context of a wrapped function. The run-time system performs a type-specific initialisation of a watchpoint's descriptor when a watchpoint is added to a pointer/address with a known type. Type-specific descriptor initialisation is a powerful feature because it allows for new watchpoints to be lazily added (thus propagating our detection capabilities) to newly discovered objects referenced *within* an object of a known type. We automatically propagate type-based watchpoints using type specifications generated from parsing C header files.

Granary recursively follows argument and return value pointers based on object type specifications. These specifications tell Granary how to follow pointers and what–if any–pointers should be converted to watchpoints. To reduce programmer effort, Granary automatically generates type specifications and wrapper functions for any set of known types and functions. Some manual post-processing is required if the analysis/debugging application needs to understand semantic relationships between function arguments or structure fields.

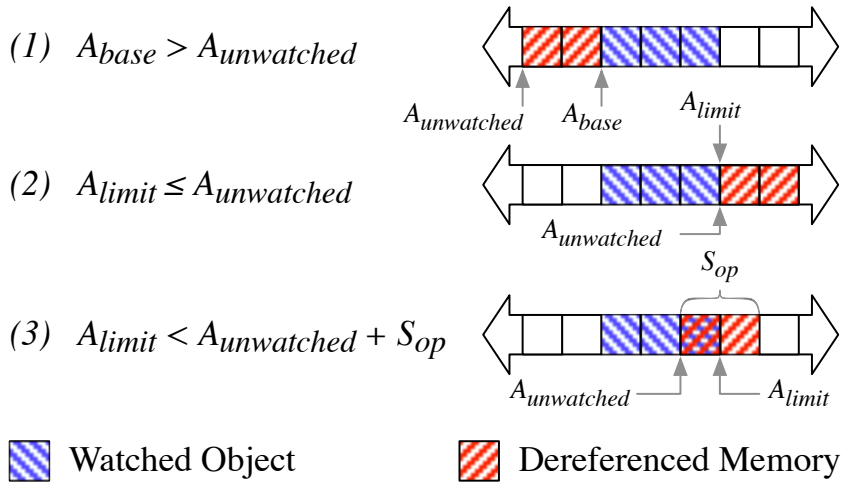


Figure 4.2: Three common buffer overflow cases are illustrated above: 1) underflow 2) overflow, and 3) overlap. Beside each illustration is the policy that detects the specific bug. When a watched address ($A_{watched}$) is dereferenced, a vtable function that is specific to the memory operand size (S_{op}) is invoked. This function detects a buffer overflow if the intended referenced memory address ($A_{unwatched}$) does not fall within the boundaries delineated by the base and the limit addresses (A_{base} and A_{limit}, respectively).

4.1.3 Stack-based overflow detection

Detecting stack overflows is important because they are commonly used in remote code execution and privilege-escalation attacks against operating systems [43]. We detect overflows on stack allocated objects when memory outside of the bounds of the activation/call frame in which the object is allocated is accessed. This policy is distinguished from the heap- and type-based policies in that the watchpoint descriptor is managed separately from the watched memory, and the lifetime of the descriptor extends beyond the lifetime of the watched memory.

To detect stack overflows, we view the memory occupied by the activation frame of an invoked function as a dynamically-sized buffer. Like our other buffer overflow policies, we use a watchpoint to detect accesses to memory outside of this buffer. Unlike our other policies, we only *associate* a descriptor with this buffer, and rely on a different mechanism to *add* the watchpoint to stack addresses. We separate adding watchpoints from allocating descriptors for stack overflow detection. In particular, 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

Benchmark	Native	Watchpoint Null	Everything watched	Buffer overflow
Microbenchmark	1x	2.7x	3.8x	20x

Table 4.1: The overhead of watchpoint instrumentation for buffer overflow detector on microbenchmark shown in Figure 3.3. The buffer overflow detector uses instrumentation for bounds checking on the access of watched objects.

before the address is dereferenced.

4.1.4 Evaluation

We evaluated the performance of our buffer-overflow detector using a the synthetic microbenchmark shown in Figure 3.3 and measured the performance of common filesystem operations on an in-memory disk using the *iozone* filesystem benchmark. We used the same experimental setup as described in Section 3.4. The overhead of the buffer-overflow detector on synthetic microbenchmark is shown in Table 4.1. The high overhead of detector is due to the instrumentation needed for precise bounds checking. Figure 4.3 shows the overhead of the buffer-overflow detector on the throughput of common filesystem operations issued by the *iozone* file system benchmark. We saw ~80% drop in the throughput of file system operations. Much of these overhead comes because of code centric instrumentation and the overflow detector watches all the module allocated objects. Watching selected objects for buffer overflow detection will improve its performance.

4.2 Selective Memory Shadowing

Shadow memory enables a program analysis tool to track the history of every memory location and/or value in memory. Existing tools that use shadow memory such as Memcheck [28],

TaintCheck [31], TaintTrace [9], and Eraser [39] demonstrate that shadow memory is a powerful and widely used method for developing program debugging applications.

Memcheck [28] uses shadow memory to track what allocation/deallocation operations have affected each memory location, and can thus detect accesses to unaddressable memory. It also tracks the undefined memory locations (uninitialised or derived from undefined locations) and can therefore detect dangerous uses of undefined memory. TaintCheck [31] and TaintTrace [9] are security tools. They track which values are from untrusted (tainted) sources and which values were subsequently derived from them, and can thus detect dangerous uses of tainted values. Eraser [39] uses shadow memory to track which locks are held when a memory location is accessed, and can thus detect when a memory location is accessed without a consistent lock-set, which may imply a data race. In the above applications, shadow memory provides the ability to detect critical errors, which otherwise could go undetected.

The implementation of shadow memory in these tools is inherently expensive because all memory needs to be shadowed [29]. The tools need to maintain extra state for each byte of memory and instrument all loads and stores to keep this state up-to-date. These requirements increase the total amount of code that is run and the program's memory footprint, thus significantly affecting performance. Instead, behavioral watchpoints enable implementing a novel shadowing scheme that allows tracking objects selectively, which decreases the amount of code that needs to be run and the program's memory footprint.

We implement selective shadow memory by maintaining a watchpoint descriptor-specific, variable-sized bitset. Each byte of allocated memory corresponds to one bit of descriptor-specific shadow memory. The bits in shadow memory are initialised to zero. Individual bits are flipped to one by write-specific vtable functions or read-specific vtable functions when a write/read operation happens to the bytes of memory shadowed by those bits.

We use this shadow memory scheme to detect memory error bugs (e.g., uses of uninitialised memory, memory freeing bugs for heap-allocated objects) and to characterize the types of objects accessed by different kernel modules. As described in Section 4.1.1, Granary interposes on the kernel's memory allocators and adds watchpoints to the addresses returned by those allocators.

When memory is allocated, the watchpoint descriptor is initialised with the number of allocated bytes (available as an argument to the allocator) and with its own shadow memory.

Uninitialised memory is defined as either allocated memory to which no value has been written, or deallocated memory. We selectively track the initialisation state of each byte of watched memory using *shadow memory* [28]. Our implementation represents shadow memory as a bitset, where the state of each bit tracks the initialisation state of a byte of watched memory. Watchpoint descriptors are augmented to include the size (in bytes) of a watched object and a pointer to the memory shadowing the object[1].

4.2.1 Read-before-write bugs

Read-specific vtable functions detect 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.

4.2.2 Memory freeing bugs

We detect freeing of non-heap memory when the argument to a kernel memory deallocator is not a watched address. We detect an invalid free, such as freeing an offset of an allocated object, when the unwatched address of a watched object being deallocated is different from the watchpoint descriptor's base address. 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

[1] As an optimisation, shadow memory for small objects can be embedded in the watchpoint descriptor.

bug. We detect double-free bugs when the descriptor of a watched address being deallocated is already marked as dead.

Two positive side-effects of our approach is that it detects double-free bugs and freeing of non-heap memory bugs. A double-free is detected when a watched address for an already deallocated object is passed as an argument to a deallocator. An attempt to free non-heap memory is detected when an unwatched address is passed as an argument to a deallocator.

4.2.3 Fine-grained access pattern

Kernel modules execute in the kernel address space with unbounded privileges. These modules are implicitly trusted and interact with each other and the kernel through well defined interfaces and by sharing data in an uncontrolled manner. Unfortunately, the assumed trust leaves commodity OSes vulnerable to malicious and misbehaving kernel extensions[8, 24, 47]. We detect the access patterns of the kernel modules using selective shadowing. Behavioral watchpoints tracks the objects of interest and any read/write access to these objects by both the kernel or the module code is captured by the shadow memory. The shadow memory information is logged at every control transfer from the kernel to the module. This information is useful for characterizing the access patterns of kernel modules, when may then be used to classify the module to detect module vulnerabilities.

4.3 Memory Leak Detection

A memory leak occurs when a program does not free dynamically allocated memory or when memory can no longer be referenced. Memory leaks are especially a problem in long running programs, and in operating systems, where memory is lost until the next boot. Existing tools such as Purify [38], Valgrind [41] and Dr Memory [5] provide an effective solution for detecting memory leaks in user-space programs but they are not available for the kernel. Other systems such

as *kmemleak* [10] and SystemTap [13] provide support for debugging memory leaks in the kernel but they operate on the kernel as a whole and not on specific modules of interest.

Implementing a leak detector only for kernel modules is challenging for three reasons.

First, a leak detector for kernel modules should follow all objects allocated in the module context. Many of these objects should be deallocated by the module. For example, a file system module may make a call to `create_workqueue` and allocate a `work_struct` that can be used to schedule the delayed work. It is the responsibility of the module to deallocate the object before exit. Keeping track of such objects for a leak detector operating on the entire kernel, such as *kmemleak*, is easy since they track all the memory allocation and deallocation but it is challenging for a leak detector operating only on the kernel module.

Second, a leak detector needs to perform reachability analysis on all objects allocated in the module context to detect the lost references. Existing leak detectors use different garbage collector algorithm such as reference counting and tracing-based algorithm to detect the live objects. Performing such analysis in only module context is challenging because module regularly lose internal references to the objects that they allocate. For example, a network module can allocate objects and pass with the `net_device` structure to the kernel, without retaining references to those objects. The network module can later indirectly access these objects using the kernel's `netdev_priv` interface. A whole kernel leak detector, such as *kmemleak*, operating on kernel heaps & static data can detect such lost references by scanning all allocated objects but a module based leak detector, having limited knowledge of objects allocated in the module context, can not detect the references of such objects and will report them as a leak.

Third, the existing leak detector in the kernel needs to trace all the memory allocator and deallocator functions in order to track the allocated objects. It also needs to trace the `container_of` macros to get the addresses generated from a pre-allocated objects. *kmemleak*, being tightly integrated with the kernel source code, performs this by interposing at the memory allocators/ deallocator (e.g, `kmalloc`, `vmalloc`, `kmem_cache_alloc` and other friend functions) and tracing the `container_of` operations on pre-allocated objects. Performing such operations by a leak detector, having no knowledge of the kernel activities and operating only on the module binaries is

not possible.

We implemented the leak detector for the kernel modules using the watchpoint framework. The challenges involved in implementing a leak detector only for the kernel modules make behavioral watchpoints a suitable solution. Behavioral watchpoints solve the challenges involved in three ways:

i) The watchpoint framework ensures that all objects allocated in the module context are watched. These watched addresses are easily disambiguated from normal memory addresses and integers that look like watched addresses. This is done by taking over all kernel memory allocators and tracking the execution context of the module running under its control. This helps the framework identify the context of every memory allocation and adds watchpoints on all these objects. However, in the process the watchpoint framework may add watchpoints on the objects not related to the module. Adding extra watchpoints on objects does not affect the correctness of the leak detector. We detect such objects during the collector scan and suppress the false positive that may get reported due to them.

ii) Behavioral watchpoint solves the problem of losing internal reference of objects allocated in the module context by providing leak detector complete visibility of the watched objects. Any read/write accesses performed on these watched objects get detected by the watchpoint instrumentation or the hardware traps which attaches the instrumentation framework with running code. The read/ write operations on watched addresses mark these objects live and the collector scan considers these objects as reachable. The ability to track the accessed objects is particularly helpful in detecting stale objects that are potentially leaked. The leak detector based on garbage collector algorithm such as *kmemleak* can not detect such objects.

iii) Behavioral watchpoints help associate the allocated object with the descriptors. The descriptors hold meta-information for each objects and help identifying the state of a memory block without much of the effort. This helps leak detector limit the scope of scanning to objects that are allocated and live. The behavioral watchpoints are also viral and every addresses

derived from a watched addresses (e.g., through copying or offsetting) is also watched. The watchpoint framework does not need to trace the pointer arithmetic used to derive addresses from the allocated objects. *kmemleak* reports false positive if the program uses other than `container_of` operations to generate derived addresses.

4.3.1 Object Liveness Analysis

The effectiveness of a leak detector in identifying the object leaks depend on the "accuracy" of the liveness analysis. An ideal leak detector identifies all heap-allocated objects that are not dynamically live and reports them as leak. A dynamically-live heap object is one that can be reached by following pointers that may be dereferenced in the future.

Existing leak detector solutions such as *kmemleak* [10], Purify [38] and Dr Memory [5] use a garbage collection algorithm to perform object liveness analysis. They use the mark/sweep collector algorithm [3] to detect reachability of the allocated objects starting from a set of "root-set" objects. Their root-set includes system registers and non-heap addressable memory.

They perform reachability analysis periodically at intermediate position and it involves stopping the client when collector[2] is in process. However, the solution is not very efficient when client operates on a large number of heap-allocated objects, such as is typical in operating systems. It stops the client for longer time.

One solution for reducing the pause time is to use generational or parallel collection. Generational collection avoids stop-the-world by running the collector thread occasionally but it doesn't remove the problem completely. However the parallel collectors take an orthogonal approach and reduce the time of pause by running the collector in parallel to the client.

In our implementation of leak detector we used the parallel collector to perform object liveness analysis. Keeping in view of the challenges involved in performing object liveness analysis, we

[2] A low priority daemon thread which runs intermittently in the background performing reachability analysis on the allocated objects

used three different policies to perform the leak scan.

Scanning of accessed objects: This is an aggressive scanning policy for detecting memory leaks. The approach is similar to bookmarking collectors [18] and Melt [4] which track the accesses of allocated memory blocks to detect stale objects at page or object granularity. The approach considers all objects that are accessed between two consecutive scans as live objects and performs reachability analysis considering them as "root-sets". Any object which is not reachable from these objects is considered as stale or a potentially leaked object.

The leak detector tracks all accesses of the watched objects and updates the meta-information associated with their descriptors. The scanner thread (collector) periodically scans the descriptors stored in a global descriptor table and marks the objects that were accessed in the last epoch as live. The scanner thread later performs reachability analysis considering these objects as "root-sets". The leak detector also tracks the sources of memory allocation in the descriptors and a constant increase in the stale objects from one source is reported a possible leak.

This approach reduces the overhead of leak scan which is performed by the collector periodically and it is also helpful in finding stale objects (dead but reachable) which the existing garbage collector algorithm based leak detector can not detect. However, the approach performs limited reachability analysis, thus resulting in increase in the false positive especially for objects which does not get accessed for longer interval of time. It also can't find the objects with live leak i.e the allocated objects are getting accessed periodically but these accesses are nonetheless useless. Detecting such leaks with any leak detector is challenging.

Scanning of module reachable objects: We define module reachable objects as objects which a module can access directly or has accessed in the past. It includes static data for both the kernel and the modules, system registers, threads stack executing module code and all heap-allocated memory viewed by the module directly or indirectly. Module viewed objects are the objects allocated by the kernel but passed to the module during its course of execution. We keep track of such objects

by interposing at the kernel/module interface and collecting all the kernel objects passed to the modules. The type information about these objects helps us in associating a type-specific scanner with each object, which performs deep scanning of these objects during a leak scan.

In this leak scan policy, the collector considers all module reachable objects as the "root-set" and recursively scans them to perform liveness analysis. This scan policy uses the mostly-parallel collector [3, 2] instead of the parallel collector for updating the "root-set" before performing the leak scan. This is important to have a consistent snapshot of system registers and thread stack running the module code. The parallel scanning of kernel objects requires maintaining consistency and avoid the cyclic dependency of dependencies across data structures. We detect cyclic dependencies by hashing each memory block during a scan and storing them in a data structure. The scan of a memory block is stopped if its hash is found. We also use the scan depth as a fall back mechanism to avoid walking a long list of objects in a data structure. In our implementation we used a scan depth of "6" which we find is appropriate for detecting all watched objects in any kernel data structure. We are also using relaxed consistency policy during the scan. This is because our leak detector makes a decision about a memory block leak after multiple scans.

This approach reduces false positives compared to the first approach because it includes module-viewed objects in the root set. However, it has a higher cost because it requires maintaining bookkeeping information for the module-viewed objects and tracking their deallocation. The approach does not remove false positives completely as it is not able to track references to objects that are allocated by the module but passed to the kernel, after which the kernel uses internal data structures to maintain references to the objects.

Scanning of kernel heaps and static data: This is the most conservative policy and used as the fall back approach for performing a leak scan. This policy scans the static data for both kernel and the modules and kernel heaps to find any references to watched objects. A watched object that is not found during the scan is definitely considered a leak.

For scanning kernel pages, we used the kernel page tables and the kernel data structure `init_level4_pgt`

to access kernel pages. The scanner thread (collector) divides the kernel pages based on the memory regions and types and scans them based on priority. The scanner thread stops scanning the kernel pages once it finds at-least one reference to watched objects.

This approach avoids any false positives but scanning the kernel heap memory is expensive. Also, the scanning of kernel pages can cause false negatives because leaked objects may have internal pointers containing watched addresses. We reduced the problem by removing the memory block of watched objects that are not live from the scan list.

Types of Leaks The three scanning policies for performing liveness analysis on watched objects detect leaks at different levels. Based on the leaks reported by these policies, the detector divides allocated memory objects into three categories:

i) Live objects: These objects have been accessed between leak scans and are thus live and reachable from one of the roots. Objects are marked live after the scanning of accessed objects

ii) Stale objects: These objects have not been accessed for a small number of scans but there is a possibility that they are live. Objects are marked as stale after the scanning of module reachable objects. The decision about these objects are made after performing the full system scan of kernel heaps and static data.

iii) Garbage objects: Objects that are not reachable and are certainly garbage. The decision about such objects is made after scanning the kernel heaps and static data of the kernel and its module.

Meta-information The leak detector maintains two kinds of meta-information to track the state of a memory block and to detect leaked objects. This meta-information is added in the descriptor of an object.

Benchmark	Native	Watchpoint Null	Everything watched	Leak detector
Microbechmark	1x	2.7x	3.8x	12x

Table 4.2: The overhead of watchpoint instrumentation for the leak detector on our microbechmark in shown Figure 3.3. The leak detector uses instrumentation to mark live objects between the scans.

- *leak_state :* This includes the state of the memory block. It tracks if the memory is allocated and is accessed in the last epoch. It also counts the number of epochs since the object not been accessed. Our current implementation of the leak detector is aggressive and it performs a full system scan for an object if it does not get accessed in the last two epochs.

- *memory_info :* This includes basic information about a memory blocks such as base address, size and type and source of allocation.

4.3.2 Evaluation

We evaluated the performance of the leak detector with the synthetic microbenchmark (Figure 3.3) and the macrobenchmark discussed in Section 3.4. The overhead of the watchpoint instrumentation for the leak detector is shown in Table 4.2. The leak detector adds extra instrumentation to mark objects as having been accessed (i.e live) between the two scans. This adds overhead to the watchpoint instrumentation.

Section 4.3.1 discusses three scan policies for implementing the leak scan. We also evaluated the cost of each of the scanning policies. Table 4.3 shows the overhead of the three scan policies in terms of the CPU cycles needed to complete the scan. The cost of scanning module reachable objects depends on the number and size of objects the collector needs to scan.

We evaluated the effectiveness of each scanning policies in detecting leaks by introducing memory leak in the kernel module. We removed calls to deallocator functions (`kfree` and

	Descriptors scan	Module reachable objects	Kernel heaps & static data
CPU cycles	591822	3000269	18859648171

Table 4.3: The cost of different scanning policies for the collector. The cost of scanning module reachable objects depends on the number and size of the objects in hash table. At the time of evaluation the hash table was having "192" objects with the total scan size of ~4Mb.

`kmem_cache_free`) from the `ext3` file system module and we consider all the objects that should be deallocated by `ext3` module as leaked. We used same experimental setup as discussed in Section 3.4 and mounted `ext3` module on a RAMDisk of size 1 GB. We used the macrobenchmark with system utilities `cp` and `tar` as the workload which was operating on a source tree of `openssh-6.2`.

Scanning of accessed objects: This policy performs a leak scan only on the objects that are accessed in the last epoch. The knowledge of accessed objects in each epoch allows us to implement two different scanning sub policies. The first sub policy acts as the baseline and considers only the accessed objects as live. It is aggressive in nature and may lead to many false positives.

The second sub policy considers all accessed object as live and include them in the "root-set" to perform reachability analysis on watched objects. It reduces the number of false positives by performing shallow scan on these live objects.

Table 4.4 shows the effect of both these sub policies in detecting leaks. It shows there is an increase in false positives with sub policy 1. This is because of its aggressive nature in detecting memory leaks. This reported leak is valid for a kind of workload and may vary with different workloads.

We also performed the staleness analysis on the allocated objects using this policy. We used an aggressive policy in which objects that are not accessed between two epochs are considered stale.

sub-policy 1: Accessed objects are live but are not "root-sets"						
scan	Allocated objects	Accessed objects	Reachable objects		Unreachable objects	
			Not Leaked	Leaked	Leaked	Not Leaked
1	25	24	13	11	0	0
2	305	240	145	95	35	30
3	2053	1545	23	1522	480	28
4	2069	38	23	15	2010	21

sub-policy 2: Accessed objects are live and added into "root-sets" for reachability analysis						
1	25	24	13	11	0	0
2	540	370	165	283	88	4
3	2129	1120	40	1221	862	6
4	2314	39	24	318	1968	3

Table 4.4: The profile of the allocated objects and detected memory leaks when the scanner thread (collector) considers the accessed objects in last epoch as live and performs reachability analysis. The reachable objects which are not leaked is less than the accessed object in each epoch. This is because many of the allocated objects were transient and should have deallocated before the next scan starts.

scan	Allocated objects	Reachable objects		Unreachable objects	
		Not Leaked	Leaked	Leaked	Not Leaked
1	27	19	0	8	0
2	705	65	11	629	8
3	1426	125	11	1290	18
4	1824	221	10	1584	9

Table 4.5: The profile of the allocated objects and detected memory leaks when the scanner thread (collector) uses module reachable objects as "root-set" and performs reachability analysis.

The stale objects are potentially leaked and there is need to perform full system scans if the number of stale objects increases constantly. Figure 4.4 shows an increasing number of stale objects with an increase in the allocated objects. This is because of our aggressive policy of detecting stale objects. The number of these stale objects also depends on the type of workload used for analysis. In this case we used workload which compile the source tree of `openssh-6.2` and copy the source tree of `gcc-4.8` on RAMDisk mounting `ext3` module.

Scanning of module reachable objects: The policy considers all objects which are reachable from the module as "root-set". We used the same evaluation strategy for this policy and all memory deallocator functions from the `ext3` file system module are removed. Table 4.5 shows that the policy detects actual leaks with less false positives and negatives.

Scanning of kernel heaps and static data: This is the fall back policy for scanning and is not aggressive in nature. Table 4.6 shows that the policy reports a lot of false negative. However, it is fairly accurate in detecting leaks and an objects not found during this scan is definitely a leak.

scan	Allocated objects	Reachable objects		Unreachable objects	
		Not Leaked	Leaked	Leaked	Not Leaked
1	59	52	28	7	0
2	789	30	734	25	0
3	805	30	305	470	0
4	1623	24	797	802	0

Table 4.6: The profile of the allocated objects and detected memory leaks when the scanner thread (collector) scans all the kernel allocated pages.

compare with *kmemleak* We also compared the performance of our leak detector with the overhead incurred by *kmemleak*, a leak detector which operates on the whole kernel. We used server benchmarks from Filebench to evaluate performance. Figure 4.5 shows that the performance of our leak detector is comparable to *kmemleak* but our system has overhead only when the module is used where *kmemleak* has overhead on the entire kernel. Also our leak scan policies are customizable and use fall back policy only when there is a constant increase in stale objects. *kmemleak* always uses the fall back mechanism of scanning which is costly.

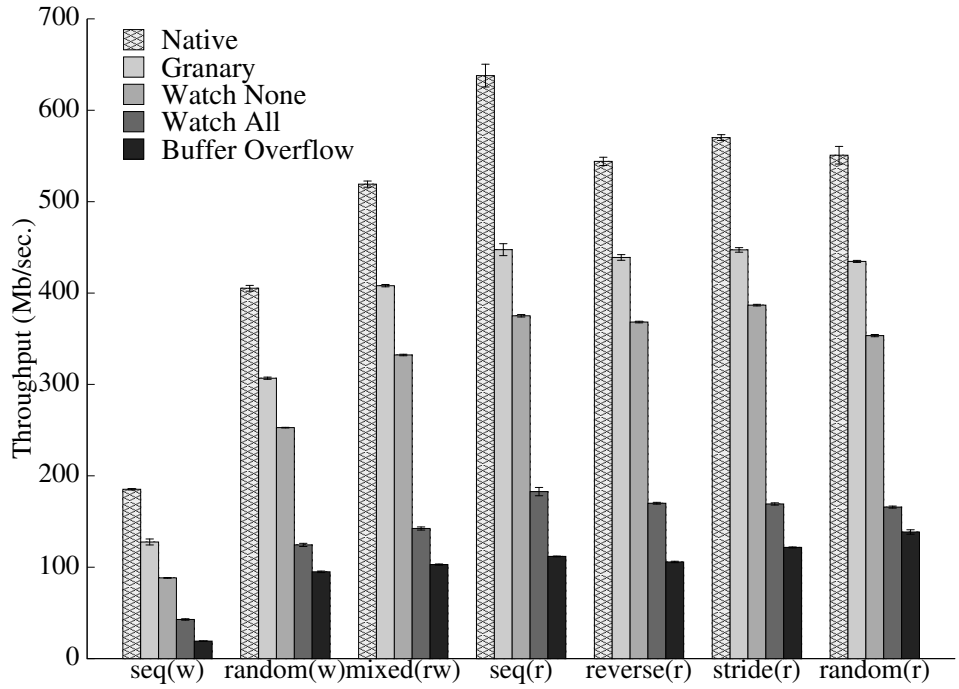


Figure 4.3: Performance overhead (Throughput in MB/sec of common file system operations) of buffer overflow detector when used to detect the buffer overflow in `ext3` file system module. The buffer overflow detector uses code centric instrumentation for the module code and data centric instrumentation for the kernel code. The overflow detector watches all the memory allocated by the `ext3` and `jdb` modules.

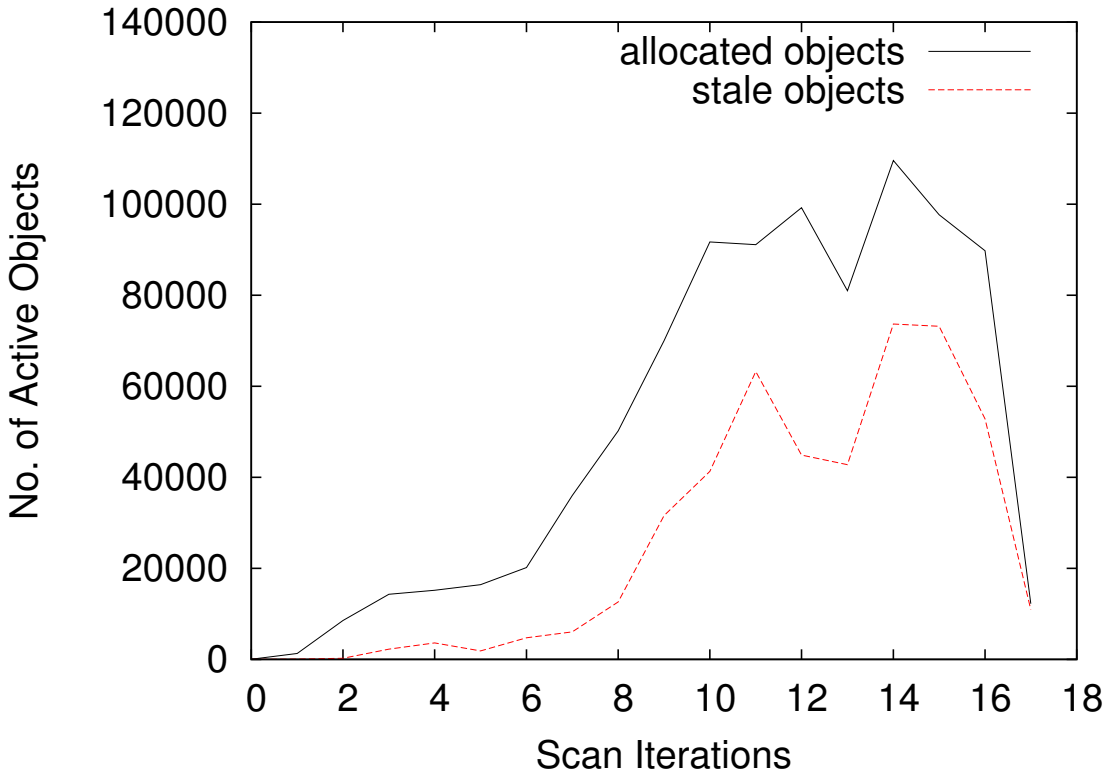


Figure 4.4: The number of stale objects with increase in the number of allocated objects. The scanner thread was scheduled after every 100ms and the workload ran for 17 scan iterations.

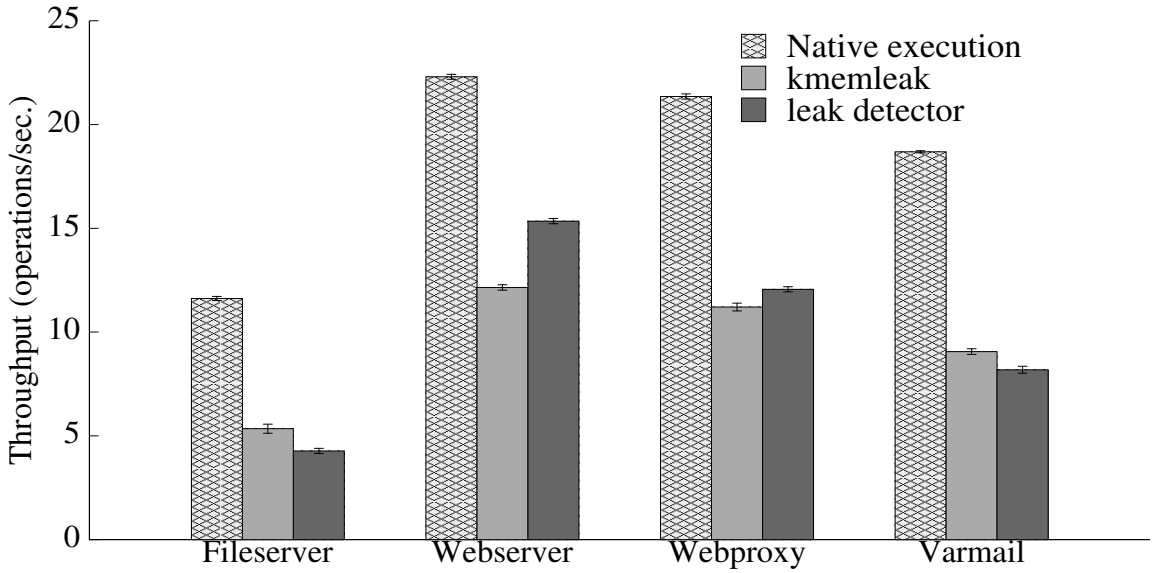


Figure 4.5: The performance overhead of our leak detector when using the Filebench benchmark. It also compares the overhead of leak detector with *kmemleak*, which operates on the entire kernel.

Chapter 5

Related Work

In this chapter, we discuss research topics that are related to our work. We first focus on the different solutions of implementing software and hardware watchpoints. We later look into the existing applications that can be used to detect memory errors.

5.1 Watchpoints

Watchpoints are an important debugging facility that help users identify data corruption bugs. There have been several proposals in the past on how to implement hardware and software watchpoints.

Greathouse *et al.* [17] propose a hardware solution that efficiently supports an unlimited number of watchpoints. Watchpoints are stored in main memory and utilize two different on-chip cache: a bitmap and a range cache, to accelerate performance. The bitmap lookaside buffer stores fine-grained watchpoints while the range cache can efficiently hold large contiguous regions of watchpoints and is useful for setting watchpoints on ranges of memory. Witchel and Asanovic [46]

describe the implementation of the Mondriaan (also spelled Mondrian) memory protection domain system for the Linux kernel. Mondriaan was designed with fine-grained inter-process protection mechanism in mind, and is optimized for applications that do not perform frequent updates. The protection information is stored in main memory and is cached in a protection lookaside buffer (PLB). The implementation of protection domains using hardware enables fine- and coarse-grained memory protection, the mechanism which is similar to hardware watchpoints.

Zhou *et al.* [49] introduce intelligent Watcher (also known as iWatcher), which provides architectural support for monitoring program execution with minimum overhead. iWatcher stores per-word watchpoints alongside the cache lines that contain the watched data. These bits are initially set by hardware and are temporarily stored into a victim table on cache evictions. The hardware falls back to virtual memory watchpoints if this table overflows. iWatcher can watch a small number of ranges, which must be pinned in physical memory. If this range hardware overflows, the system falls back to setting a large number of per-word watchpoints. In general, this system is inadequate for tools that require more than a small number of large ranges.

Unlike behavioral watchpoints, these approaches depend on specialised hardware and require that applications using this hardware maintain context-specific information separately.

Suh *et al.* [43] propose a method of secure program execution by tracking dynamic information flow. Their method uses a one-bit tag to indicate whether the corresponding data block is *authentic* or *spurious*. The scheme can be extended to use multiple-bit tags if there are many types or sources of data. 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.* [48] describe a method for implementing an efficient and scalable DBT-based watchpoint system. It uses on-demand based dynamic instrumentation to accelerate the software debugger and supports more than a million watchpoints. It uses page protection and indirection through a hash table to track watched memory. This approach does not supports watching ranges of memory, or context-specific information.

Lueck *et al.* [23] introduce semantic watchpoints as part of the PinADX system, an extension

of the PIN DBT framework. 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.

Wahbe *et al.* [45] also proposes the implementation of software watchpoint using code patching and static analysis. Another interesting approach, proposed by Keppel. *et al.* [20] is to use checkpoints for memory updates. However all these approaches are valid for userspace programs only. The latest patches for MemCheck in Valgrind [42] also introduce support of adding watchpoint. They use data structure to maintain the watchlist and track the usage of watchpoints. This puts a severe restriction on their performance.

5.2 Memory Debugging

There are many software, hardware or hybrid approaches proposed for detecting memory bugs. Purify [38] and MemCheck [29] are two widely used software tools for detecting memory usage problems. Purify was one of the first tool to combine detection of memory leaks with detection of use-after-free errors. Purify uses a technique called object code insertion (OCI) to insert checking code around every instruction in an application that references, allocates, and deallocates memory. It uses a memory-coloring scheme to keep track of the state of every byte of memory in the application. The link time instrumentation and reporting reads of uninitialized errors immediately result in false positives. MemCheck [29] is build on the Valgrind dynamic instrumentation framework. Dr. Memory is another memory checking tool developed using the DynamoRio instrumentation system. Both Memcheck and Dr Memory use a similar approach for tracking memory usage and both of them use Purifys basic leak detection approach to detect memory leak errors.

The memory debugging tools developed using behavioral watchpoints also maintain the state of a memory block in its descriptor. The descriptor information gets encoded with the memory address and is directly available when the address is accessed.

Parallel Inspector [35] is a commercial tool built on the Pin [7] dynamic instrumentation plat-

form that combines data race detection with memory checking. But the details about its implementation are not publicly available. Insure++ [33] is another commercial memory checking tool. It supports inserting instrumentation at various points, including in the source code prior to compile time, at link time, and at runtime, but its more advanced features require source code instrumentation. Insure++ also uses DBT-system to insert instrumentation at runtime. Unlike Insure++, Behavioral watchpoints does not require source code support for detecting memory errors. It uses a technique called reifying instrumentation to provide the benefits of high-level static analysis information in specialising instruction level instrumentation.

There have been several proposals to extend current hardware support for debugging memory related errors. DISE [11] (Dynamic Instruction Steam Editing) is a general hardware mechanism for interactive debugging. It adds dynamic instructions for checking memory references into the instruction stream during execution. Other solutions such as SafeMem [37], iWatcher [49], and MemTracker [44] have also been proposed for detecting inappropriate usage of memory with low overhead. Unlike behavioral watchpoints, they require specialised hardware to implement their solutions.

Chapter 6

Conclusion and Future Work

Bibliography

[1] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, Chi-Keung Luk, G. Lyons, H. Patil, and A. Tal. Analyzing Parallel Programs with Pin. *Computer*, 43(3):34–41, March 2010.

[2] Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Program. Lang. Syst.*, 27(6):1097–1146, November 2005.

[3] Hans-J. Boehm, Alan J. Demers, and Scott 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.

[4] Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 109–126, New York, NY, USA, 2008. ACM.

[5] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, Washington, DC, USA, 2011. IEEE Computer Society.

[6] Derek L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Cambridge, MA, USA, 2004.

[7] Prashanth P. Bungale and Chi-Keung Luk. Pinos: a programmable framework for whole-system dynamic instrumentation. In *In VEE*, 2007.

[8] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast Byte-Granularity Software Fault Isolation. In *SOSP*, 2009.

[9] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications*, ISCC '06, pages 749–754, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Corbet. Detecting kernel memory leaks. `http://lwn.net/Articles/187979/`, 2006.

[11] Marc L. Corliss, E Christopher Lewis, and Amir Roth. Low-overhead interactive debugging via dynamic instrumentation with dise. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 303–314, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 133–147, New York, NY, USA, 2005. ACM.

[13] Frank C. EIgler, Vara Prasad, Will Cohen, Hien Nguyen, Martin Hunt, Jim Keniston, and Brad Chen. Architecture of systemtap: a linux trace/probe tool, 2005.

[14] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive Kernel Instrumentation via Dynamic Binary Translation. In *ASPLOS*, 2012.

[15] Bryan Ford and Russ Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

[16] Peter Goodman, Akshay Kumar, Angela Demke Brown, and Ashvin Goel. Granary: Comprehensive kernel module instrumentation. Poster at OSDI'12, 2012.

[17] Joseph L. Greathouse, Hongyi Xin, Yixin Luo, and Todd M. Austin. A case for unlimited watchpoints. In *ASPLOS*, 2012.

[18] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 143–153, New York, NY, USA, 2005. ACM.

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

[20] David Keppel. Fast data breakpoints. Technical Report UWCSE 93-04-06, University of Washington Department of Computer Science and Engineering, April 1993.

[21] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[22] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. Jinn: synthesizing dynamic bug detectors for foreign language interfaces. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 36–49, New York, NY, USA, 2010. ACM.

[23] Gregory Lueck, Harish Patil, and Cristiano Pereira. PinADX: an interface for customizable debugging with dynamic instrumentation. In *CGO*, 2012.

[24] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *SOSP*, 2011.

[25] Barton P. Miller and Andrew R. Bernat. Anywhere, any time binary instrumentation. In *ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Szeged Hungary, September 2011.

[26] Arndt Müehlenfeld and Franz 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.

[27] Robert Muth. Register liveness analysis of executable code. Technical report, 1998.

[28] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *VEE*, 2007.

[29] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[30] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[31] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the*

Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA. The Internet Society, 2005.

[32] W. Norcott and D. Capps. IOzone file system benchmark. URL: www.iozone.org.

[33] PARASOFT Corporation. Insure++, 2000. Disponvel em ftp://ftp.parasoft.com/insure++/ins_win.exe.

[34] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 2–11, New York, NY, USA, 2010. ACM.

[35] Paul Petersen. Intelparallel inspector. 2011.

[36] Mark Probst, Andreas Krall, and Bernhard Scholz. Register liveness analysis for optimizing dynamic binary translation. In *Ninth Working Conference on Reverse Engineering (WCRE02*, pages 35–44, 2002.

[37] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.

[38] Er Ro Rs. Purify: Fast detection of memory leaks and access errors.

[39] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[40] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC 2012*, 2012.

[41] Julian Seward and Nicholas 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.

[42] Julian Seward and Nicholas 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.

[43] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[44] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 273–284, Washington, DC, USA, 2007. IEEE Computer Society.

[45] Robert Wahbe. Efficient data breakpoints. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, ASPLOS-V, pages 200–212, New York, NY, USA, 1992. ACM.

[46] Emmett Witchel, Junghwan Rhee, and Krste Asanovic. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *SOSP*, pages 31–44, 2005.

[47] Haizhi Xu, Wenliang Du, and Steve J. Chapin. Detecting exploit code execution in loadable kernel modules. In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 101–110, Washington, DC, USA, 2004. IEEE Computer Society.

[48] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. How to do a million watchpoints: efficient debugging using dynamic instrumentation. In *CC / ETAPS*, 2008.

[49] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iwatcher: Efficient architectural support for software debugging. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 224–, Washington, DC, USA, 2004. IEEE Computer Society.