Capability Wrangling Made Easy: Debugging on a Microkernel with Valgrind

Aaron Pohle Björn Döbel Michael Roitzsch Hermann Härtig

Technische Universität Dresden, Germany
{apohle,doebel,mroi,haertig}@os.inf.tu-dresden.de

Abstract

Not all operating systems are created equal. Contrasting traditional monolithic kernels, there is a class of systems called microkernels more prevalent in embedded systems like cellphones, chip cards or real-time controllers. These kernels offer an abstraction very different from the classical POSIX interface. The resulting unfamiliarity for programmers complicates development and debugging. Valgrind is a well-known debugging tool that virtualizes execution to perform dynamic binary analysis. However, it assumes to run on a POSIX-like kernel and closely interacts with the system to control execution. In this paper we analyze how to adapt Valgrind to a non-POSIX environment and describe our port to the Fiasco.OC microkernel. Additionally, we analyze bug classes that are indigenous to capability systems and show how Valgrind's flexibility can be leveraged to create custom debugging tools detecting these errors.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]; D.2.7 [*Distribution, Maintenance, and Enhancement*]: Portability

General Terms Design, Reliability

Keywords Valgrind, Microkernel, L4, Capability

1. Introduction

The growing complexity of modern software systems causes a growing number of errors [3, 16] both during development and on systems shipped to end-users. This problem calls for tools to aid developers in finding bugs during production and to deploy higher quality code. Many such tools are available for commodity systems, we however focus on a different kind of system here. Our group develops the Fiasco.OC microkernel and the L4Re runtime environment [15], forming a system following the L4-line of microkernels [17].

Microkernels follow the system design philosophy that the kernel provides only mechanisms, but no policy. A concept is only allowed in the kernel if security or performance reasons demand it. Such systems offer interesting properties such as high security through strong isolation, real-time guarantees, and flexibility through fine-grained object composition. However, these systems also come with new problems for developers. Managing capabilities for resource access and delegation, communicating with components by mapping memory pages, and handling page faults in user space are challenging tasks for programmers used to the convenience of the POSIX interface. These difficulties should be met with new debugging tools for developers.

Existing debugging methods for commodity systems include formal verification [4, 13, 31], static analysis tools [6, 7] and dynamic analysis tools [18, 22]. The latter group observes the program's actual behavior at runtime and is thus capable of detecting problems that arise because of resource constraints, timing influences, user input, or interaction with the underlying platform or other components. We found this tool category to be well-suited for debugging system-level components on a microkernel. Dynamic analysis tools can only check code paths that are visited during test runs and therefore fail to identify problems in code that is not executed. This is a general limitation of dynamic analysis and not specific to our work.

These tools are tightly integrated with the platform requiring them to be adapted to our system. We chose the Valgrind dynamic binary analysis framework [22] for our work because of its wide applicability and source code availability.

Implementing dynamic analysis in POSIX-compliant environments such as Linux is well understood [18, 22] and has lead to the development of valuable tools [20, 24]. This paper describes our experiences in making the Valgrind framework available in the context of our microkernel-based operating system, which comprises Fiasco.OC and L4Re [15]. This paper makes the following contributions:

1. We explore design alternatives when porting Valgrind to a microkernel operating system interface. For that, we need to look beyond the POSIX nomenclature to see what functionality Valgrind actually needs. We show how microkernel mechanisms, such as capabilities, local name spaces and interposition, support Valgrind's operation on such systems. We believe designers of other non-POSIX systems can benefit from our results by learning how their specific platform features might suit Valgrind. To the best of our knowledge, our work is the first to deal with running Valgrind on a non-POSIX operating system.
2. We provide a detailed look at memory management. Valgrind assumes that memory is managed transparently by the kernel. However, in our microkernel system, management of an application's virtual address space is implemented inside user-level

http://doi.acm.org/10.1145/1735997.1736001

VEE'10, March 17–19, 2010, Pittsburgh, Pennsylvania, USA.

applications. This leads to pitfalls where critical Valgrind assumptions about applications' execution behavior are violated. We solve this problem by adding another layer of virtualization between Valgrind and the application.

3. Fiasco.OC uses capabilities as means for resource management and delegation. Programmers' unfamiliarity with this concept leads to programming errors. We analyze such errors and present a Valgrind-based tool, *CapCheck*, that helps us to analyze problems specific to capability management.

We start with a brief description of our port's prerequisites: the L4Re microkernel runtime environment and the Valgrind dynamic analysis tool (Section 2). We continue with details on porting Valgrind (Section 3). CapCheck, a tool for debugging problems specific to capability systems, is introduced (Section 4). We provide a performance evaluation (Section 5) and conclude after discussing related work (Section 6).

2. Existing Technologies

In this section, we provide the reader with a common understanding of the two existing code bases involved: the Valgrind dynamic analysis tool and the L4Re runtime environment. We explain details only to the extent needed to comprehend later parts of the paper. More extensive knowledge can be found in separate literature: The L4Re part is largely based on [15] and microkernel design principles explained in [17]. For more details on Valgrind, see [22].

2.1 Valgrind

Valgrind is itself not a debugging tool, but rather a binary translation and instrumentation framework for building debugging tools. Such instruments can perform analysis or profiling tasks. Existing tools like *MemCheck* detect bugs in memory management including buffer overruns, use of uninitialized memory or double-frees. Problems such as data races which occur in multi-threaded applications can be detected with the *helgrind* tool. Further, there are tools for cache-profiling (*cachegrind*), call-graph-generation (*callgrind*), or profiling of memory-usage (*massif*). If no tool matches the analysis use case, it is rather straightforward to write a new one on top of the Valgrind framework.

The program being debugged is called the Valgrind *client* in this paper as this name is used by the Valgrind authors in publications and source code. However, since Valgrind's operation is related to virtualization some publications also speak of Valgrind *guests* which is synonymous.

How Valgrind Works

To achieve its flexibility, Valgrind uses dynamic binary translation and execution to perform the tool-specific instrumentations. During runtime, Valgrind and the client share the same address space. Valgrind inserts itself as a virtualization layer underneath the client and employs binary translation to achieve total control over client execution.

Valgrind works at basic-block granularity. Every block is translated into a Valgrind-specific intermediate language. Translated blocks are handed to the currently active tool for modification. The tool can insert any instrumentation it desires. When the block is handed back to the Valgrind framework, the instrumented intermediate language block is recompiled into native code. Valgrind then executes it, making sure to receive control back at the end of the basic block's execution. To accelerate execution, previously recompiled basic blocks are cached to avoid repeated compilation of the same client code block. To simplify tool development, Valgrind also allows interposition of C-style function calls and system calls. Tools can register handler code for such events.

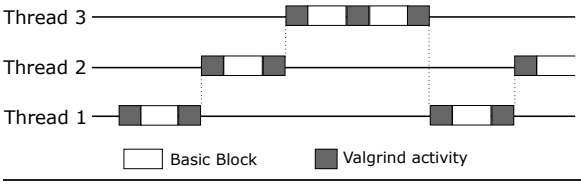


Figure 1. Serialization of a multi-threaded application in Valgrind.

Multi-threading

To support multi-threaded applications, Valgrind employs one host thread per client thread. With Valgrind's tight control over execution, multiplexing of a single Valgrind thread to multiple client threads appears possible, but this raises problems when threads are suspended due to blocking system calls. Using multiple threads solves this problem. However, to allow the development of powerful tools like MemCheck, which use shadow-value technology [21], Valgrind must guarantee that shadow-values and actual values are always consistent. This is implemented by serializing basic block execution as shown in Figure 1. *Switching between threads only happens at basic block boundaries and must never happen within a basic block*. Synchronized by a global lock, only one thread is active at any given time. The active thread changes only at basic block boundaries. Threads conceptually change contexts between being a Valgrind thread and a client thread. In the Valgrind context, they translate and instrument basic blocks, while running as a client thread they execute the previously prepared basic blocks. We show later, how this design incurs problems on a microkernel.

2.2 Fiasco.OC and L4Re

The runtime environment and the underlying Fiasco.OC microkernel follow the classic microkernel design paradigms, augmented with an object-capability system. Like every microkernel, Fiasco.OC only offers mechanisms, but no policy. To achieve a minimal kernel, concepts are only allowed in the kernel if a user-space implementation would not meet the security needs of the system. Such inevitable concepts include address spaces, threads and communication.

The microkernel approach is very different from traditional monolithic kernels. The majority of today's commodity systems follow this monolithic alternative. In such a system, the kernel is orders of magnitudes larger in terms of source lines of code and includes many more features. File systems, drivers, networking and even rudimentary graphics support is typically part of the kernel. At their system-call layer, these kernels directly offer the richness of the POSIX interface or — in the case of Windows — something similar.

In a microkernel-based system, the aforementioned components would be implemented as user-level services and not be part of the kernel. To improve fault resilience, components live as separate server applications in different address spaces. Even seemingly fundamental functionality like memory management is performed in user space with the kernel implementing only basic page table manipulation to support memory-mapping decisions. As a result, microkernel-based systems provide stronger isolation properties due to the decoupled components.

L4Re is a user-level runtime library that simplifies management and communication by providing a framework to implement Fi-

asco.OC client and server applications. Our system also comes with a POSIX compatibility layer but this provides only an incomplete subset of the POSIX standard and relies on backend servers to implement files and networking. The native L4Re platform API remains non-POSIX-conforming and applications usually use convenient POSIX features along with interfaces provided by the base system as they please. Valgrind not only assumes a POSIX-system, but also that the POSIX interface is provided by the kernel in the form of system calls.

Objects

The kernel implements objects, such as task and thread objects, for the abstractions it provides. Objects are not exclusive to the kernel; arbitrary additional objects can be implemented in user-level. This is transparent to the caller of an object. The implementor of a user-level object defines an interface to interact with the object and creates an *IPC-Gate*, which clients can use as an entry point to communicate with the object. The IPC-Gate acts as a proxy between client and server, allowing user-level servers to reimplement and augment kernel-level objects, a mechanism commonly called interposition.

Capabilities

To address objects in our system we do not use a global naming scheme, as global names disclose the existence of other applications within the system. Instead, capabilities are used to denote objects and regulate access to them [11]. Capabilities facilitate containment, because an application cannot employ a service unless it possesses a valid capability to it. Access must be explicitly granted instead of being available by default. In a system with global names, the name of the server could be guessed, so additional access control would be needed. Capabilities also facilitate the selective delegation of rights by transferring a capability to another task.

Our system features only one system call: invoking a capability. Behind the scenes, this is of course discriminated according to the kernel object associated with the capability. Invoking a capability constitutes sending a message via the capability. Message payload is passed in the CPU registers and a dedicated messaging area in memory that acts as additional, virtual CPU registers. This area is located inside the user-level part of the thread control block (UTCB).

Capabilities are maintained by the kernel in a per-task table, called the *capability space* of a task. The entries in this table, called *capability slots*, refer to a kernel object, which in the case of an IPC-Gate object can represent any user-level-implemented object. Applications access their capabilities by stating the index into their capability table similar to the use of file descriptors in a POSIX system. Allocation and deallocation of capabilities happens at the discretion of the application.

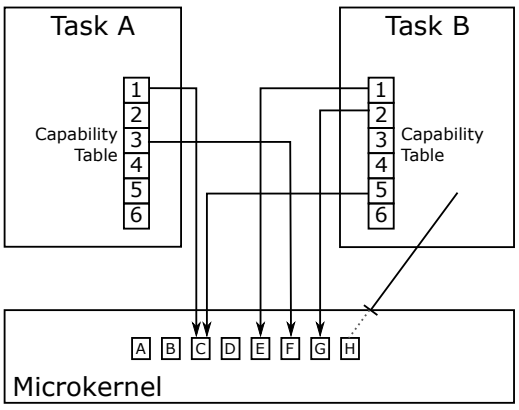


Figure 2. Object-capability mappings in Fiasco.OC

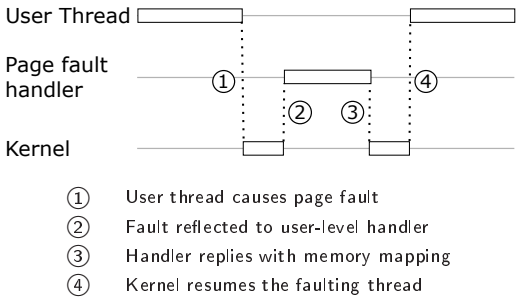


Figure 3. User-level page fault handling.

Figure 2 depicts two tasks and their capability relations. The microkernel provides a set of kernel objects that are referenced through capabilities (A-H). Each task has a local capability table (indices 1-6) to which the kernel maps certain objects. Task A can for instance access object E under the name '3'. Tasks A and B share access to object C (named '1' for A and '5' for B), which might for instance constitute a communication channel. Capability invocation being the only system call, there is no possibility for a task to gain access to any object that is not mapped into its capability table. In this example there is no way for task B to access object H because of that.

The combination of objects and capabilities as fundamental design elements of our system inspired the name Fiasco.OC with OC being short for object-capabilities.

User-level Memory Management

In a monolithic system, page faults are transparently resolved by the kernel. This is fundamentally different on a microkernel where page faults are reflected by the kernel to a user-level page fault handler. As this strongly influenced design decisions we made during our port, we cover page fault handling in more detail here.

In a monolithic kernel, a page fault causes the CPU to trap into the kernel page fault handler. This code inspects the faulting address and consults the region list which contains information on the intended memory layout of the application. From the region list, the page fault handler learns whether the missing page is supposed to come from a file, from device memory, or from plain RAM. It allocates a new page, potentially obtains the page content from a file and manipulates the page table so the page becomes visible to the application. The kernel then resumes execution of the faulting thread which causes the CPU to repeat the last instruction with the requested memory now accessible.

As shown in Figure 3, a page fault on a microkernel also causes the CPU to trap into the kernel (1). The kernel however checks which user-level thread is registered as the faulting thread's page fault handler. The kernel then reflects the page fault to this so-called pager thread (2). This pager consults a region list to learn what kind of memory is supposed to populate the faulting address. Since this region information is application-specific knowledge, the design adopted by L4Re is to have the pager be a dedicated thread within the same address space. In this design, the pager is called the *region mapper*. According to the region list, the region mapper consults a server for RAM, files or device memory to receive the needed

page. It then replies to the kernel with a page mapping resolving the page fault (3). The kernel changes the page table accordingly and resumes execution of the faulting thread (4).

Regarding page faults, a characteristic of Valgrind is that faults can occur in two contexts: Valgrind itself may cause a page fault in its own code or memory. Additionally, executing the translated and instrumented client code may cause page faults in memory used by the client. Since all Valgrind threads execute both contexts, each thread can cause both types of page faults. We will see in the next section that this becomes a problem for our port.

3. Valgrind on L4Re

This section presents details of our Valgrind port to the L4Re runtime environment. The most difficult challenge we faced was to support Fiasco.OC's notion of user-level memory management. Section 3.1 elaborates on these problems. We present possible design approaches in Section 3.2 and our final solution using capability interposition in Section 3.3. Additionally, Section 3.4 gives some information on changes we made to Valgrind apart from memory management.

3.1 Memory Management Problems

Valgrind's goal is to take any application binary and wrap its execution completely so that all program behavior can be instrumented. L4Re applications contain their own, dedicated region mapper thread, so staying true to Valgrind's principle would require to run this thread under Valgrind control as well. However, porting Valgrind to L4Re causes Valgrind itself to employ a region mapper to manage its own memory needs. Consequently, with such a design we end up with two region mapper threads in one task — one originating from Valgrind itself and one being pulled in by running an L4Re application as the client. The former would be executed natively, the latter would be translated and instrumented by Valgrind as part of the client. This design causes three problems:

Problem 1: Pager Ambiguity Any thread in Valgrind switches between executing Valgrind code and client code. This context however determines, to which of the two region mappers page faults should be forwarded to. Unfortunately, a Fiasco.OC thread can only have one associated page fault handler.

Problem 2: Conflicting Management Both region mappers are managing the same address space. The region mapper code in L4Re was developed with the assumption that it manages its address space exclusively. While we could adapt Valgrind's region mapper to remove this assumption, the second region mapper is part of client code which we want to run unmodified. The resulting double-management leads to conflicts when the client's region mapper tries to map memory to an address already used by Valgrind.

Problem 3: Concurrent Basic Blocks The third problem is a consequence of interference between microkernel memory-management and Valgrind's assumption of basic block atomicity discussed in Section 2.1. Imagine Valgrind executing a client basic block. The client code touches some unmapped memory, causing a page fault. This will suspend the thread in the middle of the basic block and divert control to the kernel. The page fault is reflected to the appropriate region mapper. Since this is a client page fault, it would have to be handled by the client's region mapper. Ignoring problems 1 and 2 for now, let us assume we had a way to perform this forwarding. This would wake up the client's region mapper, which is a Valgrind-controlled thread. If Valgrind would begin instrumenting and executing the client's region mapper, depending on the implementation it would either:

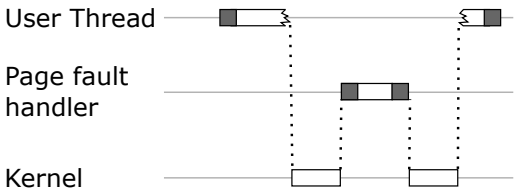


Figure 4. User-level memory management violates basic block atomicity.

a) deadlock when trying to acquire Valgrind's global lock, which is still held by the thread causing the page fault or

b) violate Valgrind's assumption that basic blocks are executed atomically, because we have a half-finished basic block suspended in the faulting thread. The resulting inconsistencies may cause unpredictable tool behavior.

This situation is illustrated in Figure 4.

3.2 Solution Space

The pager ambiguity problem can be solved by modifying Valgrind to change the active pager of a thread whenever switching between Valgrind and client execution contexts. This incurs an overhead of one extra system call per switch. The problems of conflicting management and basic block concurrency induce multiple possible solutions which we discuss in the following. We will come back to the ambiguity problem in the next section.

Address space layout synchronization

Valgrind needs to keep track of all changes to the client's virtual address space layout so that tools like MemCheck can adequately track memory allocation. Conversely, the client must be denied memory mappings to areas used by Valgrind. However, the address space layout is managed by the L4Re region mapper as shown in Figure 5. Thus, synchronization between the Valgrind-internal address space layout information and the client region mapper is necessary. This synchronization can be implemented by either instrumenting all system calls performed by the client, wrapping certain memory allocation functions using Valgrind's function wrapping mechanism, or by using capability interposition. We now discuss each of these approaches.

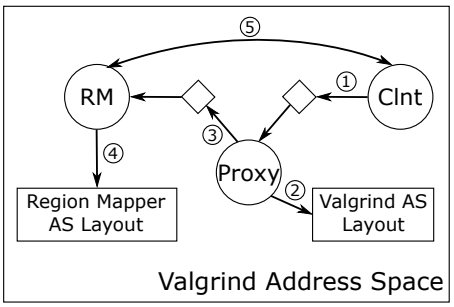
Valgrind comes with a mechanism to *intercept system calls* from the client. This mechanism provides two callback functions to be called before and after each system call. These can be used to analyze system call arguments and thereby track all changes to the virtual address space. An error can be injected should the client request a mapping to Valgrind memory. This is similar to the way Valgrind operates on a POSIX kernel: There, Valgrind intercepts the `mmap` system call. Lacking such a dedicated system call, this is an intrusive solution on Fiasco.OC. We would need to deeply inspect all system calls to ascertain whether they involve a mapping operation. This would cause a high instrumentation overhead.

A more viable mechanism provided by Valgrind is the option to *redirect function calls* from the client. By wrapping the L4Re functions used to modify the virtual address space, we can inspect all changes to the client's address space layout and adapt Valgrind's internal representation. This approach decreases the instrumentation overhead, but relies on applications to use the function names we

redirect. An application that chooses to manually contact the region mapper instead of using L4Re's C interface would not run correctly.

A third solution is depicted in Figure 6 and is similar to function wrapping, but uses Fiasco.OC's capability features instead of Valgrind's own instrumentation. For contacting its region mapper, an L4Re application uses a well-known capability slot located within its environment. When starting the client, Valgrind modifies the client's environment, so that all region mapper requests are redirected to a Valgrind-provided proxy region mapper that extracts information from client map requests to keep Valgrind's memory layout information up to date. The proxy then forwards requests to the actual region mapper of the client or injects an error if the client tries to establish a mapping in Valgrind memory. This is similar to system call interception, but allows us to specifically proxy only those requests that handle memory mappings. The capability redirection installed by Valgrind is transparent to the client, so no modifications to the binary are necessary.

These solutions address the conflicting management problem. We now discuss approaches for the problem of concurrent basic blocks before we summarize the complete solution we actually implemented.



① RM calls are redirected to a region mapper proxy
② Proxy updates Valgrind address space layout
③ Proxy redirects message to original region mapper
④ RM updates address space layout
⑤ RM handles page faults

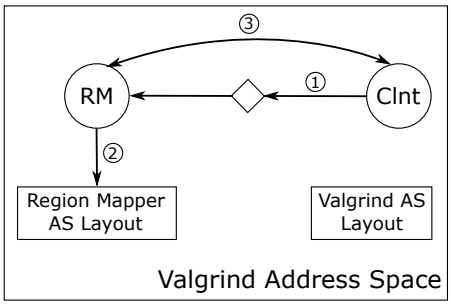
Figure 6. L4Re memory management - with proxy

Basic block atomicity

Valgrind's assumption that basic blocks are always executed atomically needs to be taken care of in the presence of user-handled page faults. The most drastic solution would be to *eliminate this assumption* from Valgrind. Currently, Valgrind only performs thread switching and the corresponding bookkeeping at basic block boundaries. The scheduler could be adapted to switch at arbitrary locations. However, this would require heavy modifications to Valgrind, so we did not pursue this any further.

Another possible modification to Valgrind would be to *allow parallel execution of threads* and thereby eliminate the internal limitation of serialized execution. This has been tried by the authors of the pValgrind [28] project. Their version of Valgrind allows simultaneous execution of multiple instrumented threads. Unfortunately, their work still has some limitations, such as not supporting Valgrind's most important tool, MemCheck, which is not suited for parallel execution.

To allow user-level paging without having multiple basic blocks in flight, we could speculatively execute each basic block under the assumption that it will not raise any faults. This will succeed most of the time and for the occasions where page faults are raised, we *rollback the executing thread* to the beginning of the basic block, and then send its pager a page fault message. After the resolution succeeded, we restart the faulting basic block. This approach would require fewer modifications to Valgrind, namely having to take a checkpoint of relevant thread and memory state before each basic block. Software transactional memory [9] could be used for the implementation. Additionally, an external thread would be needed to intercept page faults and initiate rollback.



① Client requests memory invoking region mapper capability
② RM updates address space layout
③ RM handles page faults

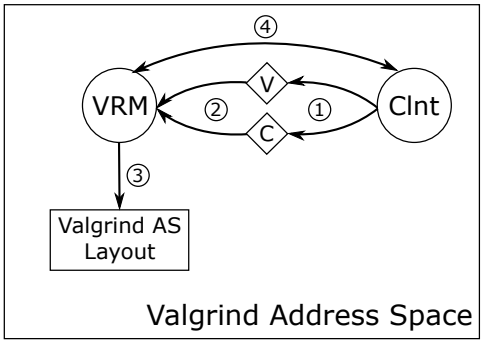
Figure 5. L4Re memory management - default case

Any such invasive modification to Valgrind is unattractive as it complicates not only the initial port but also any future upgrades to newer versions of Valgrind. Instead of modifying Valgrind to accommodate our pager architecture we could also redesign our memory management to avoid concurrent basic blocks altogether. The problem originates from the client's region mapper being setup and executed as instrumented client code. In a POSIX system however, paging happens as a kernel service transparently to the client. We can do this similarly by consolidating the client's and Valgrind's region mapper. Memory is still managed on user-level, but by Valgrind on behalf of the client. The client's region mapper becomes obsolete. Because Valgrind's region mapper is not instrumented, page faults can be resolved without running instrumented code and thus without violating Valgrind's assumption about atomic basic blocks. The drawback of circumventing the client's region mapper is that we are not able to instrument and debug its implementation. However, the region mapper only represents a tiny fraction of the code base we aim at and we are willing to accept this.

The next section describes how we consolidate the client memory management into the Valgrind region mapper and how we combine this idea with the concept of capability interposition discussed earlier.

3.3 Memory Manager Virtualization

Our port of Valgrind to L4Re consolidates user-level memory management in one region mapper thread that runs untranslated inside Valgrind's address space and intercepts all region mapping requests and page faults that are caused by client code. It also serves as the region mapper for Valgrind itself. Due to this multiplexed functionality, we call it the virtualized region mapper (VRM) thread. This design solves the problem of conflicting management, because all memory layout information is consolidated in one data struc-



① Client issues memory requests using V(algrind) or C(lient) capabilities depending on its role.
② Updates are all propagated to the VRM thread
③ VRM updates memory layout
④ VRM handles page faults

Figure 7. Virtual region mapper (VRM)

ture. This design further eliminates the problem of concurrent basic blocks, because the VRM is not run instrumented, so page faults no longer cause multiple in-flight basic blocks.

Calls to an application's region mapper are invocations on a region mapper capability that is part of the application environment. Virtualizing the region mapper capability is simplified by the design of Fiasco.OC and L4Re. During client startup, Valgrind modifies the client's environment by replacing the region mapper capability with one pointing to the VRM thread. Thereafter Valgrind uses a Fiasco.OC system call to register VRM as the client's page fault handler as well.

This way, the VRM handles all mapping requests and all page faults for both Valgrind and client. Tools such as MemCheck however need to distinguish client from Valgrind mappings to check the validity of memory accesses. The problem of context ambiguity resurfaces here: Memory requested by Valgrind and by the client must be handled differently, but the VRM will see requests from the same thread in both cases.

Our solution is illustrated in Figure 7. The VRM has two distinct entry points in the form of two IPC-Gates. Valgrind installs a capability to one of them as the region mapper capability used by the client. The other IPC-Gate is used by Valgrind itself for memory allocation. This way, based on a thread's current context — executing Valgrind code or client code — the correct capability will be used automatically and the VRM can tell client mappings from Valgrind mappings. This is transparent to both the client and Valgrind.

3.4 POSIX Mechanisms and Threading

Porting Valgrind to L4Re raised additional issues apart from memory management. Valgrind's implementation relies on a POSIX interface provided by the underlying operating system. This includes reliance on mechanisms such as pipes and support for signals. We replaced the use of pipes by way of the semaphore mechanism provided by Fiasco.OC. Signals have no direct equivalent in Fiasco.OC and are therefore completely ignored.

Two further issues arise with respect to threading. Each Fiasco.OC thread is assigned a User-Level Thread Control Block (UTCB) that is used for storing system call arguments. Now that Valgrind and the client code run within the same thread and share this UTCB area, this leads to problems. If the client issues a system call, Valgrind intercepts this call and then thrashes UTCB content by performing another system call itself. To prevent this, we provide the client with a virtual UTCB and only copy this data to the real UTCB at the point where Valgrind performs a system call on behalf of the client.

Fiasco.OC's interface for thread creation allows the user to specify the starting instruction pointer for the new thread. In order to retain full control Valgrind needs to detect the thread creation system call and modify this argument so that the new thread starts executing translated code instead of real code. Handling this situation is done similar to Valgrind's handling of Linux's `clone` system call.

4. Common Capability Quirks

While providing advantages in terms of security and robust resource management, the capability mechanisms used in Fiasco.OC and L4Re add another layer of complexity to the software running atop. We first describe in more detail how capabilities are handled in Fiasco.OC and the runtime environment. Thereafter, we describe classes of bugs we came across when implementing system-level software on top of this environment. Finally, we present *CapCheck*, a tool that has been designed to detect these types of bugs.

4.1 Capabilities and L4Re

The L4Re runtime is a set of decoupled objects implemented in different address spaces. Objects are globally designated by capabilities. Functions of these objects are executed by performing a system call, function parameters are passed through the UTCB. Capabilities can be mapped from one address space to another using kernel-provided communication primitives (IPC).

Capabilities are stored in a task-local *capability space* protected by the microkernel. A task cannot directly inspect or modify capability-space content. Instead, it can only refer to capabilities by their index within the capability space. The kernel does not perform any management of available capability slots. This is completely managed by the L4Re runtime.

When expecting to receive one or more capability mappings through IPC, the application needs to specify the slots to receive capabilities into when preparing the system call. A task can use the slot management provided by L4Re to allocate a new capability index before allocating an object and to free the index after releasing the object.

4.2 Bug Classes

Porting Valgrind to Fiasco.OC and L4Re was motivated by the need for debugging our software components. In addition to common errors covered by Valgrind tools, such as race conditions and memory allocation errors, we came across two classes of bugs that are caused by not understanding the subtleties of capability management.

Leakage A task is responsible for keeping track of available indices within its local capability space. Before gaining access to an object, the task needs to identify a capability index to map the capability to. The kernel does not perform any checks whether an index specified by the task is already filled with another capability. Keeping track of allocated/freed indices is up to the user-level implementation.

This may lead to capability slot leakage. We came across such an error in practice when implementing slab-based dynamic memory management [2]. Whenever a slab needs to grow, the L4Re memory allocator is used to add memory pages. Upon shrinking the slab, the corresponding area is released again.

```
void *grow_slab(unsigned size)
{
  int idx = cap_alloc();

  // call will allocate memory object
  // and map it to capability index idx.
  mem_area *mem = mem_alloc(size, idx);

  return mem->addr;
}

void shrink_slab(void *addr)
{
  // would return corresponding cap index
  mem_free(addr);
}
```

Listing 1. Capability leakage: Growing the slab allocates a new capability index for the memory area, shrinking the slab does not release the index.

Listing 1 shows a shortened version of the bug we encountered. Before allocating a new memory object, a new capability index is correctly allocated. However, the programmer forgot to free this index after releasing the memory object. Thereby, the application ran out of available capability slots at runtime causing out-of-capability errors even though there were in fact memory and capability slots available.

Overmap The microkernel does not prevent a task from overmapping an existing capability with a new one. An optimization for short-lived capability allocations is to integrate release of an old capability (unmap) and acquisition of a new one (map) into one operation by simply overmapping an already allocated capability index. However, if used improperly, this may lead to mysterious program behavior.

In practice, we encountered a bug in an application that overmapped a thread capability with a capability referring to a memory area. This automatically unmapped the thread capability and dropped the kernel's reference counter for this thread to 0. Therefore, the kernel deleted the thread. The resulting behavior was a thread sporadically disappearing from our system without any trace.

Additionally, programming errors may lead to invocation of invalid capabilities or invocation of a valid capability with invalid arguments.

4.3 CapCheck

Based on the experiences and bugs described above, we implemented CapCheck, a Valgrind tool to track common problems with respect to capability-object management in the L4Re runtime environment.

Capability Index Management

In order to track capability index allocation and deallocation, we use Valgrind's builtin function wrapping mechanism to wrap `cap_alloc()` and `cap_free()`, the functions responsible for managing capability slots in user space. We extended Valgrind's internal event tracking mechanism to generate new events corresponding to these functions. When the client allocates a capability index, CapCheck tracks the calling thread's current backtrace. Similar to MemCheck, this information is stored in a hash table until this capability index is deallocated. Upon deallocation we release the information stored along with the released index.

When the program terminates, we run through the list of all capability allocations that have not been released and print this information along with the stored allocation traces. This is sufficient to detect leakage of capability indices as described in Section 4.2.

Capability Mappings

There are two ways of tracking capability mappings within CapCheck. The first approach is to wrap relevant user-level protocol functions — such as `mem_alloc` and `mem_free` regarding memory pages — and store the obtained information. This leads to low instrumentation overhead , but requires a high maintenance effort, because all existing user-level protocols need to be analyzed and the respective wrappers need to be written. Each user-defined protocol needs to be incorporated into CapCheck. Therefore, this is a viable alternative only for a few well-known protocols.

A protocol-independent alternative is to obtain mapping information by inspecting thread state before and after invoking a Fiasco.OC system call. A thread willing to receive a capability mapping, needs to setup receive descriptors for all mappings inside its UTCB's buffer registers before issuing a system call. CapCheck stores a copy of a these buffer registers before the system call is issued. The pre-system call information is not sufficient for tracking mappings, because it only describes the maximum amount of mappings the caller is expecting. In order to track how many of these mappings succeeded, CapCheck also instruments the thread state after the system call when information on the number of successful mappings is available. With this information, CapCheck runs through the previously stored buffer descriptors and looks up all capability receive descriptors. For each of the capability indices, it stores a backtrace of the mapping system call along with the already obtained index allocation backtrace.

CapCheck's capability information therefore contains two backtraces: one identifying the location where the index was allocated and the second indicating where the last mapping into this capability index took place. This adds more context to the information about lost capabilities, which is printed out on program termination. Furthermore, it enables CapCheck to detect capability overmappings by checking whether a previous mapping backtrace exists before storing a newly obtained one.

By default, CapCheck warns about every mapping that overwrites a previously tracked capability mapping. This conservative approach may lead to false positives stemming from the design of Fiasco.OC's capability mechanism. While capability mappings are performed synchronously and can therefore be detected, capability revocation is performed asynchronously by another application. Detecting revocation is therefore only possible in a subset of cases, where well-known protocols are wrapped by CapCheck. Valgrind provides a mechanism to suppress a well-defined set of error reports. This is usually used to suppress errors in libraries which the developer cannot fix himself. It also suits our purpose to define a set of overmap reports that are known to be valid.

Capability Invocations

In Section 4.2 we described an error caused by overmapping a thread capability with a memory capability. This overmap can be detected with the mechanisms described above. However, it is still up to the developer to find out whether this is an invalid overmap operation compared to the other valid ones that may be reported for

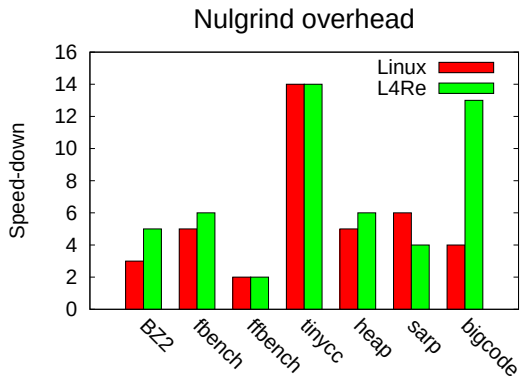


Figure 8. Speed-down for Nulgrind

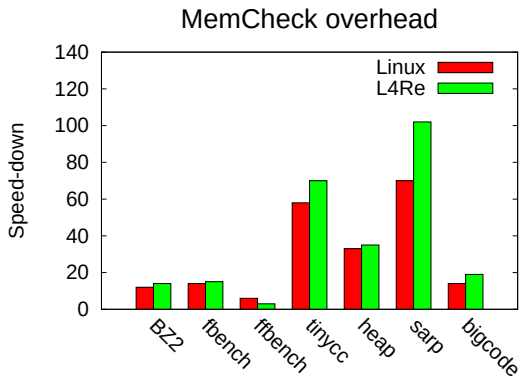


Figure 9. Speed-down for MemCheck

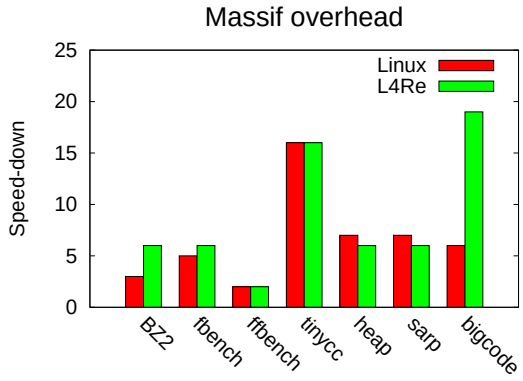


Figure 10. Speed-down for Massif

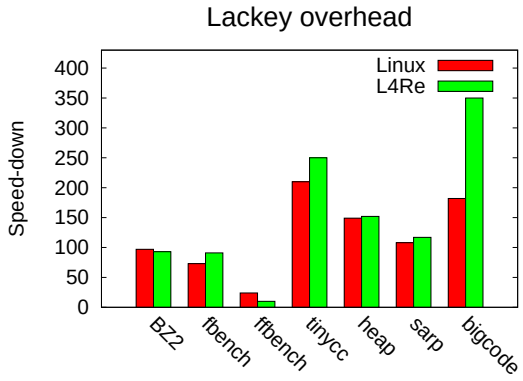


Figure 11. Speed-down for Lackey

the same application. To aid this step, CapCheck adds information on actual invocations of the overmapped capability to its reports.

The descriptor used by a thread to invoke a capability contains a field defining the user-level protocol used for this invocation. CapCheck stores this protocol within the pre-system call wrapper and checks the call's error code that is available in the post-system call wrapper. Without knowledge about the user-level protocol it is impossible to check all possible error conditions. However, we can infer valuable information in the following scenarios:

1. If a capability was invoked successfully and invocation now returns an error indicating an invalid capability, we detect that an external unmap has occurred and adapt CapCheck's data structures.
2. If a capability has been used for a well-known protocol and is now used with an incompatible protocol ID, we know that this invocation was erroneous.

Further heuristics are imaginable for inferring correct protocols used by unknown user-level objects. For instance, CapCheck could provide statistics about all protocols used for invoking a capability along with the locations in code where these protocols were used. This would enable a user to debug incorrect capability invocations. This is not implemented in CapCheck yet.

5. Evaluation

We evaluate our port of Valgrind in terms of functionality, performance and the coding effort involved.

For functional and performance evaluation, we use Valgrind's Linux variant as a reference. Instead of comparing absolute execution times, we measure the speed-down incurred by Valgrind on the respective platform. This speed-down is tool- and workload-specific and is determined by the relation between client execution time using a Valgrind tool and native execution. We expect speed-downs to be in the same order of magnitude on both platforms, with remaining deviation being attributed to differences in the underlying operating system environment.

Valgrind comes with a test suite for assessing performance and functionality. Table 1 lists these programs. We run the programs natively and with four Valgrind tools: Nulgrind — a tool only using Valgrind's translation mechanisms but not adding any instrumentation, MemCheck — a tool for detecting memory leaks, Massif — a heap profiler, and Lackey — performing binary instrumentation for the sole purpose of benchmarking.

We conduct these experiments on an Intel Celeron 2.66 GHz machine with 2 GB of RAM. On both, Linux and L4Re, we used Valgrind, version 3.4.1. Figures 8 to 11 show the obtained speed-downs for each program on both platforms.

While most of the results meet our expectations, there is one notable exception. *Bigcode* is an application copying a function to 5,000 newly allocated locations in memory and then calling these copies. This is intended to stress Valgrind's binary translator, but also raises a large amount of page faults, which are more expensive due to L4Re's pager design and our not yet optimized virtual region mapper implementation. Although a detailed performance analysis of Valgrind is not in the scope of our paper, our results show that

Benchmark	Description
bz2	Compression and decompression
fbench	Ray-tracing
ffbench	Fast Fourier transformation
tinycc	Small C compiler
heap	Heap allocation/deallocation
sarp	Stack allocation/deallocation
bigcode	Executes a lot of code

Table 1. Valgrind test suite

Unmodified Valgrind	Lines of Code
Valgrind Core	68,419
Binary translator (libVEX)	65,649
Valgrind Total	***134,068***

Table 2. Valgrind Lines of Code (Measured using David A. Wheeler's *sloccount*.)

Valgrind Adaptations	Lines of Code
Binary translator (libVEX)	13
System call handling	204
Address space management	282
VRM	416
L4Re protocol wrappers	42
CapCheck Tool	176
Various Core modifications	563
Valgrinds Scheduler	119
Startup Code needed for L4Re	459
Modified total	***2,274***

Table 3. Valgrind modifications for L4Re

Valgrind can be used as a debugging tool on L4Re with reasonable speed.

In addition to runtime overhead, we also evaluate the porting effort. Table 3 shows the number of source code lines we modified for our port and relates them to Valgrind's global code base shown in Table 2. Our modifications constitute less than 2 % of the entire code base, which we consider to be viable. We acknowledge that lines of code is not necessarily a good measure of porting effort, because the real effort can better be described by the amount of time it takes to *find* the lines of code to be modified. In this respect we found Valgrind to be well-documented and easy to comprehend.

6. Related Work

Our work aims at debugging means for system-level software components. Traditional manual debuggers such as GDB [29] are powerful tools for developer-driven bug analysis. These are the right tools to use when a bug is already known and quickly reproducible. Our work falls into the category of automated debugging tools [18, 22], which provide better support for finding bugs that either only appear after a potentially long run time or bugs that don't manifest themselves in a fail-stop manner.

With respect to operating system components, GDB has been integrated into the Linux [1] kernel and the NOVA microhypervisor [30]. These tools rely on a separate computer to be attached to the debugged system, while our Valgrind tools are running on the debugged system. Virtualization has already been used to provide debugging for a whole kernel running on the same machine [10, 12]. While this whole-system approach is useful in some scenarios, with the Valgrind approach we are able to perform more fine-grained debugging of single system components without interfering with the complete operating system.

The choice of instrumentation depends on the debugging scenarios and the overhead programmers are willing to accept when debugging. At one end of the spectrum, manual source-level instrumentation using frameworks like Ferret [27] is the best way to achieve low instrumentation overhead. However, it requires the programmer to know in advance which events to collect. Today's widespread operating systems provide semi-automated kernel instrumentation facilities either for arbitrary locations [14] or restrict this feature to pre-defined locations [19, 26, 33] in order to decrease instrumentation overhead. Valgrind's approach is to instrument binary code using binary translation. A similar approach has also been applied to instrumenting the Linux kernel by JIFL [25]. It provides full control over program execution, but has the drawback of incurring the highest instrumentation overhead.

The Valgrind instrumentation framework has proven useful for writing application- or system-specific automated debugging tools such as CapCheck presented in this paper. Other tools have shown that it can also be used for obtaining applications' system call models [8] and automatic analysis and detection of security vulnerabilities [5, 23].

7. Conclusion

We successfully ported Valgrind to the Fiasco.OC microkernel and the L4Re user-level runtime environment. Popular Valgrind tools such as the MemCheck memory leak detector and the helgrind race condition checker are therefore available for Fiasco.OC applications. Although a larger study of available L4Re applications remains to be done, we already found several real bugs, such as invalid use of uninitialized memory in VPFS [32].

We analyzed Valgrind with respect to its assumptions about the underlying POSIX-like kernel and showed necessary adaptations in order to use it on Fiasco.OC's non-POSIX interface. The main modifications were necessary because Fiasco.OC relies on user-level memory management. Our porting effort required a reasonable amount of modifications to Valgrind code.

Capabilities are a major abstraction provided by Fiasco.OC. We analyzed bugs concerning this feature and found two major types: capability leaks and capability overmappings. Using the Valgrind framework, we implemented CapCheck, a tool that is able to detect those classes of bugs.

We plan to further investigate dynamic analysis of operating system components in the future. We believe that Valgrind is a promising foundation for automatic debugging of system-level components such as device drivers. We also plan to integrate a Valgrind-like analysis framework into virtual machine monitors to aid whole system debugging and security analysis.

Acknowledgments

We would like to thank Adam Lackorzynski and Alexander Warg for their support understanding Fiasco.OC and the L4Re environment. Henning Schild, Thomas Knauth, Carsten Weinhold, Neal H. Walfield, and Christiane Berndt provided helpful comments on drafts of this paper.

This work was partially supported by the European Research Programme FP7 projects eMuCo and TECOM.

References

[1] KGDB: Linux Kernel source-level debugger. `http://kgdb.linsyssoft.com/`.

[2] BONWICK, J. The Slab allocator: An object-caching kernel memory allocator. In *USENIX Summer* (1994), pp. 87–98.

[3] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An Empirical Study of Operating Systems Errors. In *SOSP '01: Proceedings of the Eighteenth ACM symposium on Operating Systems Principles* (New York, NY, USA, 2001), ACM, pp. 73–88.

[4] CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM 50*, 5 (2003), 752–794.

[5] DREWRY, W., AND ORMANDY, T. Flayer: exposing application internals. In *WOOT '07: Proceedings of the First USENIX Workshop On Offensive Technologies* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–9.

[6] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP '01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), ACM, pp. 57–72.

[7] EVANS, D., AND LAROCHELLE, D. Improving security using extensible lightweight static analysis. *Software, IEEE 19*, 1 (2002), 42–51.

[8] FETZER, C., AND SÜSSKRAUT, M. Switchblade: enforcing dynamic personalized system call models. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems* (New York, NY, USA, 2008), ACM, pp. 273–286.

[9] HERLIHY, M., AND MOSS, J. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture* (1993), ACM, p. 300.

[10] HO, A., HAND, S., AND HARRIS, T. PDB: Pervasive Debugging With Xen. In *IEEE/ACM International Workshop on Grid Computing* (Los Alamitos, CA, USA, 2004), IEEE Computer Society, pp. 260–265.

[11] KARGER, P., AND HERBERT, A. An augmented capability architecture to support lattice security and traceability of access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy* (1984), pp. 2–12.

[12] KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Debugging operating systems with time-traveling virtual machines. In *USENIX 2005 Annual Technical Conference, General Track*, pp. 1–15.

[13] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *SOSP '09: Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (Big Sky, MT, USA, Oct. 2009), ACM, pp. 207–220.

[14] KRISHNAKUMAR, R. Kernel korner: kprobes-a kernel debugger. *Linux J. 2005*, 133 (2005), 11.

[15] LACKORZYNSKI, A., AND WARG, A. Taming Subsystems: Capabilities as Universal Resource Access Control in L4. In *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems* (Nuremberg, Germany, 2009), ACM, pp. 25–30.

[16] LI, Z., TAN, L., WANG, X., LU, S., ZHOU, Y., AND ZHAI, C. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability* (New York, NY, USA, 2006), ACM, pp. 25–33.

[17] LIEDTKE, J. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)* (Copper Mountain Resort, CO, Dec. 1995), pp. 237–250.

[18] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), ACM, pp. 190–200.

[19] MCDOUGALL, R., MAURO, J., AND GREGG, B. *Solaris performance and tools: DTrace and MDB techniques for Solaris 10 and OpenSolaris*. Sun Microsystems Press/Prentice Hall, Upper Saddle River, NJ, 2007.

[20] NETHERCOTE, N., AND SEWARD, J. How to Shadow Every Byte of Memory Used by a Program. In *VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments* (New York, NY, USA, 2007), ACM, pp. 65–74.

[21] NETHERCOTE, N., AND SEWARD, J. How to shadow every byte of memory used by a program. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments* (New York, NY, USA, 2007), ACM, pp. 65–74.

[22] NETHERCOTE, N., AND SEWARD, J. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), ACM, pp. 89–100.

[23] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS '05: Proceedings of the Network and Distributed System Security Symposium* (2005).

[24] O'CALLAHAN, R. Chronicle Recorder – Valgrind-based complete, indexed recording of process execution. `http://code.google.com/p/chronicle-recorder/`.

[25] OLSZEWSKI, M., MIERLE, K., CZAJKOWSKI, A., AND BROWN, A. D. JIT instrumentation: a novel approach to dynamically instrument operating systems. *SIGOPS Oper. Syst. Rev. 41*, 3 (2007), 3–16.

[26] PARK, I. Event Tracing for Windows: Best Practices. In *Int. CMG Conference* (2004), Computer Measurement Group, pp. 565–574.

[27] POHLACK, M., DÖBEL, B., AND LACKORZYNSKI, A. Towards Runtime Monitoring in Real-Time Systems. In *Proceedings of the Eigth Real-Time Linux Workshop* (Lanzhou, China, 2006).

[28] ROBSON, D., AND STRAZDINS, P. Parallelisation of the Valgrind Dynamic Binary Instrumentation Framework. In *ISPA '08: International Symposium on Parallel and Distributed Processing with Applications* (Los Alamitos, CA, USA, 2008), IEEE Computer Society, pp. 113–121.

[29] STALLMAN, R. M., PESCH, R. H., AND SHEBS, S. *Debugging With GDB: The GNU Source-Level Debugger*. 2002.

[30] STECKLINA, J. Remote debugging via firewire. Master's thesis, TU Dresden, 2009.

[31] TEWS, H., VÖLP, M., AND WEBER, T. Formal Memory Models for the Verification of Low-Level Operating-System Code. *Journal of Automated Reasoning - Special Issue on Operating System Verification 42*, 2 (April 2009), 189–227.

[32] WEINHOLD, C., AND HÄRTIG, H. VPFS: Building a virtual private file system with a small trusted computing base. *SIGOPS Oper. Syst. Rev. 42*, 4 (2008), 81–93.

[33] YAGHMOUR, K., AND DAGENAIS, M. R. Measuring and characterizing system behavior using kernel-level event logging. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2000), USENIX Association.