Memory Isolation as a Kernel Primitive in Dune

Adrien Ghosn
EPFL

Second Name
Second Institution

Abstract

We believe programmers would greatly benefit from an OS abstraction for memory isolation within a process. We further argue that leveraging recent architecture's hardware support for virtualizaiton would, unlike previous attempts, yield a solution that is both safe and maintainable. Hardware support for virtualization provides direct access to hardware features, such as ring protection mechanism and page tables, required to provide such an abstraction, while providing flexibility in terms of the implementation's specifics. We improve upon previous solutions by both relying on hardware mechanisms to enforce memory isolation and being completely decoupled from an existing kernel.

1 Introduction

General purpose kernels rely on the process abstraction to represent a unit of isolation in terms of memory, privilege, and execution. Processes isolate applications from each other and, more specifically, prevent one process from accessing another's memory. Although vital for operating systems, this abstraction requires process management, context switching, and resource accountability which, in turn, introduce non-negligible overheads at the OS-level, especially compared to the associated hardware costs.

We identify a discrepancy between the functionalities provided by existing general purpose kernels and user needs. While the thread abstraction provides independent, isolated, execution units within a process, memory isolation is attainable only by going through the costly creation and management of a new process. In fact, the lack of a kernel primitive to isolate memory units within the same thread of execution, pushed application developers to implement their own solutions within user space.

Memory isolation within a single thread of execution is a generalization of the ring protection mechanism, a common technique used in general purpose kernel to isolate kernel and user spaces. Mode switching is a technique that allows to perform privileged operations without having to incur the cost of a context switch. Kernel and user space isolation is enforced by efficient hardware mechanisms. For example, page table entries that belong to the kernel are marked with the supervisor bit, hence putting them out of reach of user applications.

Programming languages, and more generally user applications, have attempted to provide a memory isolation abstraction within a single thread of execution. For example, some object oriented languages like Java, allow to mark class attributes as private, i.e., the object cannot be accessed from some parts of the program. This isolation is enforced using compile and runtime checks. The common technique is therefore to rely on a user application to enforce user to user isolation, i.e., to limit access to certain memory regions. This is both inefficient, since it requires software interposition on memory accesses, and hardly implemented. User applications cannot rely on hardware to enforce isolation and, more often than not, present vulnerabilities and ways to circumvent isolation mechanisms. For example, in Java, meta-programming allows to access even the private fields.

We believe that memory isolation within a single thread of execution must be provided as a kernel primitive. Furthermore, we argue that, unlike previous implementations, it is possible to provide a maintainable and flexible solution, easy to tune to special use cases, by relying on efficient virtualization and direct access to hardware features. We implement the existing lightweight contexts abstraction[REF] (lwC) as a Dune library, and argue about the advantages of having such

a solution within Dune[REF]. This paper is organized as follows: Section[REF] presents previous implementations that strove to provide a memory isolation abstraction, and their limitations. Section [REF] presents the lwC API, as exposed in our Dune implementation. Section[REF] presents how Dune was modified to allow the implementation of the lwC abstraction. Finally, Section[REF] lists future work before the conclusion in Section[REF].

2 Shortcomings of Existing Solutions

There exists several implementations claiming to provide a memory isolation abstraction within a thread of execution. We divide these implementations into two distinct categories: user-level applications and kernel level implementations. In this section, we describe general techniques used by user-level applications solutions, and explain why we believe that this abstraction has to be provided as a kernel primitive. We then proceed with existing kernel level implementations, and expose what we consider to be important limitations that prevent them from being a stable long-term solution.

Virtual machines, hypervisors, and sandboxes implemented as user level applications share common techniques to provide security mechanisms that allow to isolate untrusted code execution and prevent it from harming both the underlying system and other isolated applications. A common security and isolation requirement in such applications is to enforce memory isolation and reduce the capabilities of running applications. As a result, such applications have to duplicate the kernel's functionalities in order to obtain fine-grained control over memory allocation and access, and hence intervene in the management of resources. On older architectures, applications such as NaCl[REF] or Vx32[REF] could rely on user access to hardware support for memory segmentation to enforce memory isolation within a thread of execution. With the advent of modern x86-64 architectures and more generally the progressive disappearance of hardware memory segmentation support, user applications had to use new techniques. User level solutions include software fault isolation, code instrumentation, instruction emulation, and binary code translation. More generally, all these software based solutions rely on some mixture of code instrumentation and software checks to enforce memory isolation. NaCl[REF], for example, was modified to replace hardware segmentation with code instrumentation. The untrusted application's code is recompiled and instrumented such that instructions that access and modify memory related registers are replaced by guarded pseudo-instructions, hence insuring that their contents are consistent throughout the execution. It further replaces all memory accesses and re-writes them in terms of the guarded registers, hence insuring that every memory operation falls into the proper address range and no untrusted application can access another's resources. NaCl's solution can be seen as a software implementation of memory segmentation. NaCl authors claim that control-flow and memory integrity are insured with an average overhead of less than 7% on x86-64. While these performances are remarkable for a solution based on code instrumentation, we believe that the recompilation step required to perform the binary translation, as well as the introduction of pseudo-instructions are a burden. Moreover, software is always subjugated to bugs, unexplored execution, and exploits. We can thus consider it less reliable than solutions relying on hardware features to enforce memory isolation. As it is the kernel's role to provide resource management and isolation, we argue that this abstraction should be provided as a collection of system calls relying on safe and efficient hardware features for memory isolation.

Containers take a higher-level approach, compared to standard hypervisors, by providing an *OS* rather than a *machine* abstraction. Protected portions of the operating system are made available to containers, while allowing them to have their own abstractions for processes and other operating system components. The container approach relies on the possibility to execute in separated spaces that isolate them from each others, as well as from the host OS. More precisely, Docker containers[REF] rely on Linux LXC to provide separated namespaces and isolate containers, while control groups are used to implement resource limiting and auditing.

While presenting an interesting approach with many use-cases in devOps, containers suffer from the same security issues as previously discussed user level applications. The isolation among containers is entirely enforced by the Linux namespace mechanism, i.e., an oversized piece of software that might contain bugs and is hence less reliable than hardware based solutions. More importantly, the popularity of Docker-like solutions spread interest to their supporting technologies, sprouting updates and extensions, each of them potentially introducing new security vulnerabilities.

Kernels are at the core of operating systems and are responsible for managing resources, among which the memory. As such, they are a perfect environment to implement and expose new abstractions for memory isolation within a process. Not only does it fit in the kernel's role, but also allows to rely on hardware features to enforce memory isolation. Others[REF] before us identified the need for a new memory isolation

abstraction, and the advantages of implementing it as a kernel primitive. The following two examples both rely on page tables and hardware features to implement their abstractions.

Wedge[REF] is a kernel library that introduces the *sthread* abstraction. *Sthread*s are described as threads with fork-like semantics. More specifically, upon creation, modifiers allow a fine-grained control over resource visibility within the newly created *sthread*. Entire memory regions can be unmapped and made unaccessible from within the *sthread*. This solution, however, does not completely decouple the memory and execution units. *Sthread*s are schedulable entities and therefore provide memory isolation at a cost that comprises *sthread*s scheduling and management. They therefore can be seen as a slightly cheaper version process abstraction.

Light-weight contexts[REF], *lwC*s for short, are implemented within the FreeBSD kernel and expose the *context*, or *lwC*, abstraction. Much like *sthread*s, users have a fine-grained control over memory mappings when creating a *context*. Entire address ranges can be shared, copied, unmapped, or made read-only. Unlike *shtread*s, however, *contexts* are orthogonal to threads, i.e., they are non-schedulable entities. A single thread of execution can switch between different *contexts*. Upon a switch, the execution state is saved within the current *lwC*. This solution completely decouples the memory and execution isolation mechanisms. However, the authors had to modify the freeBSD kernel in order to implement *lwC*s. This new feature impacts memory management and might therefore introduce new vulnerabilities in the kernel due to software bugs. As a result, one should not expect *lwC*s to be merged back inside the freeBSD's master branch. We can therefore wonder how maintainable this solution really is. In fact, extra work will be required to merge back bug fixes and updates made to the original kernel into the *lwC* branch, and thus raising the question of long-time support for the *lwC* library. Without long-time support, we doubt that *lwC*s will be adopted and used as a building block for new projects. More importantly, implementing a solution within an existing kernel requires to interact with rigid APIs and respect the kernel's implementation. From an implementation point of view, this reduces the solution's flexibility and hence might rule-out some particular implementations that would prove to be more efficient in terms of performances. For example, the *lwC*s authors had to work with freeBSD's datastructures and abstraction layers for memory management. We argue that, in a virtualized environment with hardware support, we can implement a similar solution, with equivalent performances, and more flexibility and freedom in terms of the memory hierarchy abstractions and the actual implementation of *lwC* functionalities. Moreover, our solution does not require to modify an existing kernel.

3 Leveraging hardware support for virtualization

Intel and AMD identified and answered the need for virtualization hardware support. The Intel solution, called VT-x, is a virtualization extension to the x86 ISA. AMD provides a similar extension with SVM[REF]. Instead of introducing traps for each instruction accessing privileged state and relying on a VMM for emulation, these extensions strive to execute as many instructions, privileged or not, directly in hardware. To avoid involving the VMM and hence introduce delays, the hardware maintains a shadow copy of privileged state. In Intel VT-x, the CPU provides two modes of operation: VMX root and VMX non-root. Guest OSes, i.e., virtualized environments, run in VMX non-root mode where CPU behavior is restricted. Transitions between the two modes, e.g., when the VMM needs to be involved, is managed by hardware. Transitions from non-root to root and from root to non-root are called *VM exits* and *VM entries* respectively.

Hardware support for virtualization provides direct access to hardware features, such as ring protection mechanism, page tables, TLBs, from within a virtualized environment, i.e., the guest. Such features are essential to implement and enforce efficient memory isolation within the same thread of execution, as discussed previously. Using hardware support for virtualization, we therefore have an environment that is, on one hand, independent from the host, i.e., the original kernel, and, at the same time, indistinguishable from it in terms of hardware tools made available.

While a certain form of hypervisor is still needed to interface the guest and host environments, we argue that hardware support for virtualization, as much as our own limited requirements, enable to reduce the hypervisor functionalities and complexity. By design, a hypervisor with a narrow set of features is less amenable to exploits than a standard one. Any hypervisor reduced to its core has two roles: expose functionalities, and manage a set of resources. Where standard hypervisors such as[CITE] accumulate features and functionalities, emulating a complete operating system, we believe that our needs, in order to provide memory isolation within a thread of execution, can be stripped down to a bare minimum consisting of an execution unit and some

memory. Our hypervisor is therefore solely in charge of exposing a certain amount of memory resources, i.e., virtualizing memory resources. All other functionalities are implemented within the guest, by relying on direct access to hardware features and privileged state.

Inspired by the containers, we simplify our solution by working at a higher level of abstraction than standard VMM: the *process*. As explained in Section[REF], containers identified the advantages of virtualizing the *OS* rather than the *machine*. We push this reasoning a step forward and place ourselves directly at the abstraction level within which we want to provide the memory isolation abstraction. This approach main benefit is to isolate the abstraction's implementation from the underlying kernel representation and management of processes. Unlike *lwC*, our solution is completely decoupled from an existing kernel and does not have to interact with already defined and rigid APIs. We therefore have more flexibility in terms of implementation and can, along with direct access to hardware features, tune our code to perform well for our use cases.

As was the case for existing implementations[REF], we need to enforce isolation between the guests and the underlying host. By design, the host OS sees our virtualized environment as one process. It is therefore isolated from other host processes by the host kernel itself and we need not worry about it. Isolation among guests, i.e., among memory units within the same thread of execution, requires to limit the capabilities exposed to guests. While hardware based isolation is ensured within the virtualized environment, the possibility to perform a VM exit presents a potential threat. In fact, our design maps several address spaces within the virtualized environment to a single address space the host OS. As a result, transitions to the host and system calls serviced by the host must be monitored and restricted to avoid leaking information from one guest to another. We do this by limiting the functionalities available inside the guest, i.e., disable some system calls, and by validating and sanitizing arguments to syscalls passed to the host. Although similar, in principal, to code instrumentation and interposition techniques described previously, the overhead introduced concerns only system calls that trigger a VM exit and is negligible compared to the overall cost of the transition.

4 lwC as a Dune library

Dune is a system composed of a kernel module and a user library that provide applications with direct and safe access to hardware features made available by virtualization hardware support, while preserving the host OS interfaces for processes. It differs from standard virtual machine monitors by providing a *process* rather than a *machine* abstraction. In the same way Docker blablabla.

TODO:Isolaiton between host and guest relies on two mechanisms: hardware and supervisor

The Dune kernel module is a small piece of code that can be seen as a hypervisor whose only role is to safely expose memory resources to the application. By its simplicity, the opportunities for bugs blablabla....

5 Future Work

TODO: performance evaluation. Re-implement existing applications using lwc.

6 Conclusion

7 Acknowledgments

References

Notes

[1] Remember to use endnotes, not footnotes!