CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Columbia University Academic Commons

Speculative Execution as an Operating System Service

Michael E. Locasto
Dept. of Computer Science
Columbia University
locasto@cs.columbia.edu

Angelos D. Keromytis
Dept. of Computer Science
Columbia University
angelos@cs.columbia.edu

Abstract

Software faults and vulnerabilities continue to present significant obstacles to achieving reliable and secure software. In an effort to overcome these obstacles, systems often incorporate self-monitoring and self-healing functionality. Our hypothesis is that internal monitoring is not an effective long-term strategy. However, monitoring mechanisms that are completely external lose the advantage of application-specific knowledge available to an inline monitor. To balance these tradeoffs, we present the design of VxF, an environment where both supervision and automatic remediation can take place by speculatively executing "slices" of an application. VxF introduces the concept of an *endolithic* kernel by providing execution as an operating system service: execution of a process slice takes place inside a kernel thread rather than directly on the system microprocessor.

1 Introduction

A key problem in computer security is the inability of systems to automatically protect themselves from attack. In order survive or deflect current attacks, systems need an environment where defensive operations, including remediation, can take place. Recent research on "self healing" systems attempts to address this problem.

However, it is unlikely that applications can incorporate effective self-supervision mechanisms. First, any such introspective security mechanism will be subject to attack or subversion along with the rest of the application proper. Second, there is no guarantee that the *ad hoc* collection of security mechanisms developed for individual applications will be implemented correctly or provide complete coverage. An independent, comprehensive, and general supervision mechanism would be much more coherent and maintainable.

Most current technologies for supervising or sandboxing application execution require that the entire application be inside the sandbox or virtual machine. It is our hypothesis that supervising the entire execution of a process is not necessary. Instead, we advocate a virtualization approach in which only portions of an application's execution are supervised. Reducing the amount of supervision seems like it would result in a significant performance increase for most popular applications. This paper examines the use of virtualization to abstract access to the execution of a machine language from within the operating system kernel.

1.1 Virtualization Within an OS

Virtualization is a layer of abstraction interposed between an underlying resource (often a physical device or component) and "clients" of that resource. Virtualization provides a construct that looks and behaves the same as the real or physical component but is typically implemented by a software substitute. This virtual component enables three critical capabilities: isolation, inspection, and enforcement.

Virtualization is not a new idea; it was first popularly realized in the IBM System/360. Recently, the use of virtual machines has come back into fashion in both research and industry to leverage underutilized hardware, reduce management complexity, and provide isolation.

Most approaches to OS virtualization place the virtual layer either above (*e.g.*, UML, VMWare, JVM, etc.) or below (*e.g.*, Xen) the operating system. We propose a layer of virtualization *within* the kernel (VxF) that can be selectively invoked for arbitrarily fine "slices" of a process. In this approach, the entire "guest OS" is reduced to a kernel thread that is occasionally invoked, as shown in Figure 1. VxF is complimentary to and not a replacement for current VM implementations.

VxF introduces the notion of execution as an operating system service. It provides support for a set of virtual executors (virX's) within the kernel. We call this particular organization an *endolithic kernel* ("endo-" meaning

within and "-lithic" referring to tight integration with the rest of the kernel). An endolithic kernel virtualizes the CPU within a kernel thread for a portion of a process's execution. Although the main motivation for creating VxF is to provide an environment in which self-healing and automatic repair can take place, the framework can be leveraged for more than just security; we discuss some other applications in Section 3.

1.2 Motivation and Goals

Our motivation originates from our work on constructing an emulator (STEM) [23] to supervise program execution in response to exploits and errors. Unfortunately, the use of an emulator imposes a considerable performance overhead since every program instruction is executed in software. One way to ease this burden is to limit the scope of emulation to portions of the program suspected of being vulnerable, or to distribute the monitoring task among a large collection of machines [13]. In addition, our current emulator, STEM, does not follow execution into the kernel; when a system call is invoked, STEM relinquishes control to the kernel, temporarily ending supervision and protection until the system call returns. VxF can be used to help address these shortcomings.

Most self-healing and automatic reaction mechanisms follow what we term the ROAR (Recognize, Orient, Adapt, Respond) workflow. These systems (a) *Recognize* a threat or attack has occurred, (b) *Orient* the system to this threat by analyzing it, (c) *Adapt* to the threat by constructing appropriate fixes or changes in state, and finally (d) *Respond* to the threat by verifying and deploying those adaptations.

One way in which to gain enough time to execute the ROAR workflow is to "delegate and wait" by combining micro-sandboxing with speculative execution of potentially vulnerable slices of a process. If this *micro-speculation* succeeds, then the results are committed. If not, then the temporary results are ignored or replaced according to the particular response strategy being employed. Of course, knowing how long to wait is not a decidable problem[1]. We are performing related work [14] on a survey of the length of this window size for various applications.

1.3 Contributions and Organization

The major contribution of VxF is to add a policy-driven layer of indirection to the operating system to intercept and examine the actions of a process before they become "committed" or visible at the architectural level. This mechanism is accomplished by performing virtualization of a process's execution within a kernel thread. An analog to this approach at the OS level is system call interposition [7, 25, 19], which is the basis of many sandboxing techniques. These approaches differ from VxF primarily because they only seek to detect or contain the damage rather than provide any way to fix the underlying fault or vulnerability. In addition, VxF's main operation does not perform system call interposition. Instead, virX's supervise the execution of a process's instruction stream[2].

This paper presents a feasibility study; we introduce the notion of an endolithic kernel, illustrate the basic design concept, and report on our prototype implementation of VxF for the 2.6.15.6 kernel in Section 1 and Section 3. We focus on the *mechanism* of VxF – discussion of the design and construction of the *policy* layer is deferred to future work. In order to provide context for VxF's design decisions, we next consider related work on virtualization and self-healing software systems.

2 Related Work

Virtual machine emulation of operating systems or processor architectures to provide a sandboxed environment is an active area of research [1, 10, 8]. As an interesting twist, King *et al.* [9] have recently proposed using VMMs to implement rootkits. Our micro-speculation techinque is akin to approaches [21, 18] that utilize a secondary host machine as a sandbox or instrumented honeypot: work is offloaded to this host, thus minimizing exposure to the primary host.

2.1 Speculative Execution

Speculative execution is a technique used in microprocessors to execute the instructions in a code branch before the evaluation of the branch conditional is finished. Micro-speculation introduces an additional layer of speculative execution in which the acceptance of a particular execution path is not based on the evaluation of a branch conditional, but rather a higher-order constraint.

Several recent efforts make use of speculation in a number of interesting ways. Work that is closely related to ours is Oplinger and Lam's proposal [17] for using thread-level speculation (TLS) to improve software reliability. The key idea is to execute an application's monitoring code in parallel with the primary computation and roll back the computation "transaction" depending on the results of the monitoring code. Chang and Gibson [5] speculatively execute an application's code during otherwise idle cycles in order to discover targets of future read operations. Similarly, Nightingale *et al.* [16] discuss ways for performing speculative execution at the file system level in order to overcome delays in network-mounted file systems. Finally, the Pulse system uses speculation to detect and break deadlocks [12].

2.2 Recovery and Repair

Effective remediation strategies remain a challenge. The traditional response of protection mechanisms has been to terminate the attacked process. This approach is unappealing for a variety of reasons; to wit, the loss of accumlated state is an overarching concern. Furthermore, crashing leaves systems susceptible to the original fault upon restart. More elegant approaches include failure oblivious computing [22], STEM's error virtualization [23], DIRA's rollback of memory updates [24], crash-only software [4], and data structure repair [6].

The key idea of the Rx system [20] is to checkpoint the execution of a process in anticipation of system errors. When an error is encountered, execution is rolled back and replayed, but with the process's environment changed in a way that does not violate the API's its code expects. This procedure is repeated with different environment alterations until execution proceeds past the detected error point. Rx is a clever attempt to avoid the semantically incorrect fixes of failure oblivious computing [22] and error virtualization [23].

3 Virtual eXecution Framework Design

Since merely inserting emulator code into the kernel isn't likely to be easily maintainable or extensible, we propose a framework for providing this service. An endolithic kernel views the set of available CPUs as a dynamic collection where members join and leave as part of regular operation.

While some multiprocessor systems support hot pluggable CPUs, a key idea of VxF (besides implementing this capability for COTS operating systems) is that a virX that registers with VxF need not be hardware. Furthermore, a virX doesn't necessarily need to interpret or emulate the execution of machine code. This approach enables a more general response mechanism than mere software emulation. Such tasks can include delegating work to a remote CPU (*i.e.*, RPC/RMI at the instruction level), collecting data for performance tuning, or even providing a different micro-architecture. Tasks can also include a wide variety of security monitoring (*e.g.*, virus detection, DRM, host-based anomaly detection, taint-tracking [15], Secure Return Address Stack (SRAS) [11], or Instruction Set Randomization (ISR) [2]).

3.1 Overall Design

There are a few degrees of freedom to consider when designing VxF, since it represents one particular vector in the design space of automatic intrusion defense systems. VxF's design goals include being minimally invasive for applications and supporting legacy software – in order to take advantage of VxF, an application should not have to be recompiled. However, VxF may provide a means for exporting control and information to any applications that explicitly want to take advantage of it.

VxF adopts an endolithic kernel. Whereas VMM's host multiple guest OS's on a single hypervisor, an endolithic kernel "hosts" multiple processes on multiple virtual execution engines within the operating system itself. As depicted in Figure 1, a process executes normally until it requests (or is placed by some monitor in response to a signal) to be scheduled on a virtual CPU. We currently implement entry and exit into and from a virX as a system call[3] (discussed further in Section 3.4). Signals that trigger the invocation of this system call may include alerts or alarms from intrusion detection systems.

VxF changes the kernel to allow the addition of virX's (virtual processors) as loadable kernel modules (LKM). Encapsulating virX's as LKM's helps to achieve clean (un)loading semantics. Therefore, the changes made by VxF should include enough infrastructure that writing a virX as an LKM should be straightforward.

Once a new virX is loaded, processes can be scheduled for execution on it. When a process is micro-sandboxed, it is the virX that is actually executing on the hardware as a kernel thread in supervisor mode. Interesting future work would allow any virX to run on another virX (*i.e.* self-hosting). We are currently adapting the `x86` dynamic translator QEMU [3] to be a virX. Other possible virX's can include our STEM system, which is based on Bochs, Bochs itself, or Valgrind.

3.2 Limitations

The two most significant challenges for VxF center on *when* supervision should be invoked and for *how long* this supervision should last. The second obstacle involves determining the scope of supervision. Even though VxF could run continuously, many applications (especially interactive ones, or those working in a power-constrained environment) may wish to avoid the overhead associated with constant monitoring.

The second difficulty with automatic supervision is that attacks and faults can be relatively rare events, and sandboxing the application for its entire execution would needlessly impact the normal operation of the software. This observation is one of the motivations behind micro-sandboxing. However, VxF still needs a policy mechanism to indicate when the micro-sandbox should be engaged. Enabling the enforcement of such policy is fairly straightforward. Entering the micro-sandbox can be driven by asynchronous events such as alerts from an IDS. Alternatively, the invocation of the micro-sandbox can occur at well-known places in the application's execution (such as function entry or entry into a new scope).

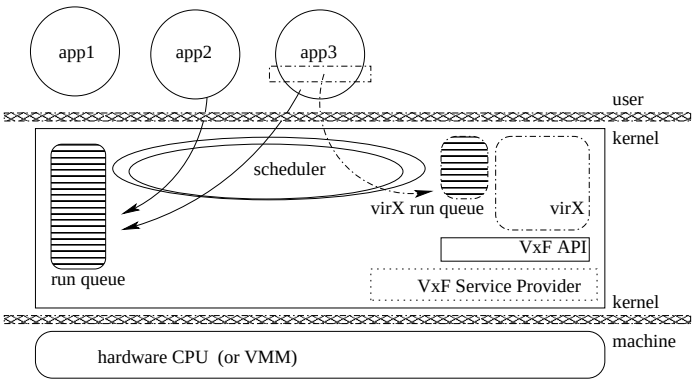

Figure 1: *Overall Design of VxF*. VxF consists of two primary components: a wrapper for the scheduler and a service provider. The purpose of the scheduler wrapper is to intercept regular scheduling of processes and redirect them to an appropriate virX runqueue. The framework's API defines a namespace and specifies contracts for a a set of services needed by virtual executors (virX's). The service provider implements the API and supports the operation of multiple virX's. In this example, `app3` normally executes on the hardware; however, a portion of it is micro-sandboxed. As a result, when execution reaches this slice, `app3` is scheduled on the virX rather than the hardware. The virX runs as a kernel thread.

However, instead of blithly executing the micro-sandbox, a small prologue can check whether policy requires that portion of the code to be supervised at that time, or in the particular state or environment configuration. Therefore, having the policy mechanism remember a snapshot of the application state and environment when attacks *do* occur is useful as evidence in the future. Follow-on work could build a system that learns when environment conditions are ripe for attack. If those conditions are recreated, then the policy mechanism could direct the micro-sandbox to kick in when it reaches the instrumentation point.

Virtualization, interpretation, and emulation all incur relatively hefty performance overhead. VxF is meant to supervise only portions of a process's execution – naturally, *not* emulating the entire execution reduces the performance penalty. However, making VxF as fast as possible is not a design goal. A clear design and implementation is favored over complexity aimed at extracting the last drop of performance from the system, since highly tuned systems are usually brittle. The properties that virtualization offers can be leveraged for security, and we assume that users are willing to justify the cost of virtualization with the capabilities afforded by these properties.

3.3 Design Alternatives

There are a number of methods for building somewhat equivalent functionality. First, we can use `ptrace()` to intercept every assembly instruction, and pass control to a handler in the kernel or in user space. Such an approach is akin to how debuggers work. We want to avoid servicing an interrupt for each machine instruction, preferring instead that the kernel remain in control of execution rather than handing control back to the hardware, being signaled, taking control back, performing some work, handing control back, etc. Second, we could take advantage of Linux's "personality" infrastructure: Linux can recognize the "type" of a file and associate an execution action with it such that another program is given the responsbility for running the "executable." Finally, rather than adding a system call for virX entry and exit operations, we could add a device to the OS that manages control signals to VxF.

3.4 Implementation

VxF has two primary responsibilities. The first is to maintain a dynamic collection of virXs by supplying functionality that includes (de)registration services, entry and exit mechanisms, and an interface that encapsulates common functionality needed by each virX. The second task is to support the ability to schedule processes for execution on a virX rather than the hardware.

VxF includes a service provider component that addresses the first task, and a wrapper to the scheduler that manages the affinity between a micro-sandboxed process and the virX it runs on. The service provider defines a namespace for virX types. The virX namespace has four parts: the namespace qualifier (currently "virx"), the prefix (an organization-specific identifier), the common name (a descriptive identifier of the virX itself), and the version. Common names describe the functionality of the virX and typically include two parts: the CPU architecture that the virX implements and a descriptive string that summarizes any additional modifications to the core execution. Examples of common names include

x86-native, *x86-stem*, *x86-qemu*, and *sparc-native*.

The straightforward approach to providing a microsandbox enter/exit signaling mechanism is to implement a new system call. Inserting this system call into the source code of existing applications violates our transparency requirements, but allows new applications to take direct advantage of VxF. Legacy applications can use a monitor that invokes the system call on behalf of the monitored application. Alternatively, we could change the OS program loader to automatically insert such calls at various places in the executable. We do not adopt this latter approach, preferring instead to cleanly separate process creation from process supervision.

VxF creates a runqueue for each virX that is loaded. Entering and exiting a virX is accomplished via a new system call, `virtexec()`. The system call doesn't immediately execute the named process on the virX; rather, it moves the process from its current runqueue to the runqueue for the appropriate virX. The modified schedular then round-robins between the multiple runqueues, giving the illusion of a multi-processor system while retaining the semantics of a uniprocessor environment. Scheduling is therefore $O(n)$ in the number of virX's loaded on the system, but otherwise retain $O(1)$ semantics of the multi-level feedback queues for each CPU/virX.

4 Future Work

Virtualization imposes a performance penalty. In order to justify this cost, we must first discover what it is. We plan to assess QEMU's performance (both standalone and as part of VxF) using the SPEC CPU2000 benchmark. We expect that its standalone performance will not be degraded by moving to the kernel.

Automatic remediation is a hard problem. Detection mechanisms are not perfect, and initiating an automated response based on a false positive is undesirable. Most remediation strategies usually result in self-induced DoS. Automating a response strategy is difficult, as it is often unclear what a program should do in response to an error or attack. A response system is forced to anticipate the intent of the programmer, even if that intent was not well expressed.

Even if automatic response capabilities existed, system security is often a matter of *policy*; systems need flexibility to remain useful in a variety of evolving environments. While VxF provides an environment for micro-sandboxing and micro-speculation, it requires some detection mechanism to trigger it and a remediation component to direct its actions when a fault occurs. We leave as future work the implementation of such a policy framework. Finally, it is of value to port VxF from Linux to other operating systems (*e.g.*, Windows, OpenBSD, or Solaris 10).

5 Conclusions

VxF is free software, and it is available at our website[4]. We welcome any comments, suggestions, or bug reports.

The ability for computing systems to autonomously detect and correct faults and vulnerabilities would improve their stability and security. The ability to execute this ROAR workflow is predicated on having an environment where supervision, detection, and repair can take place. To support this goal, VxF introduces the concept of execution as an operating system service by implementing an *endolithic* kernel organization. Such microsandboxing can be used to speculatively execute code that may contain faults or vulnerabilities.

There is no silver bullet for system security, and although it can be leveraged for more than just security, VxF is not meant to be a panacea. We advocate modifying general-purpose operating systems to (a) provide implicit supervision of instruction stream execution, (b) export a policy-driven interface for that supervision, and (c) provide the foundation for an automatic response capability via speculative execution within this supervision environment.

References

[1] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In 19th *ACM Symposium on Operating Systems Principles (SOSP)* (October 2003).

[2] BARRANTES, E. G., ACKLEY, D. H., FORREST, S., PALMER, T. S., STEFANOVIC, D., AND ZOVI, D. D. Randomized Instruction Set Emulation to Distrupt Binary Code Injection Attacks. In *Proceedings of the* 10th *ACM Conference on Computer and Communications Security (CCS)* (October 2003).

[3] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference, FREENIX Track* (April 2005), pp. 41–46.

[4] CANDEA, G., AND FOX, A. Crash-Only Software. In *Proceedings of the* 9th *Workshop on Hot Topics in Operating Systems (HOTOS-IX)* (May 2003).

[5] CHANG, F., AND GIBSON, G. Automatic I/O Hint Generation through Speculative Execution. In *Proceedings of the* 3rd *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (February 1999).

[6] DEMSKY, B., AND RINARD, M. C. Automatic Data Structure Repair for Self-Healing Systems. In *Proceedings of the* 1st *Workshop on Algorithms and Architectures for Self-Managing Systems* (June 2003).

[7] FRASER, T., BADGER, L., AND FELDMAN, M. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy* (1999).

[8] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In 10th *ISOC Symposium on Network and Distributed Systems Security (SNDSS)* (February 2003).

[9] KING, S. T., CHEN, P. M., WANG, Y.-M., VERBOWSKI, C., WANG, H. J., AND LORCH, J. R. SubVirt: Implementing Malware with Virtual Machines. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2006).

[10] KING, S. T., DUNLAP, G., AND CHEN, P. Operating System Support for Virtual Machines. In *Proceedings of the General Track: USENIX Annual Technical Conference* (June 2003).

[11] LEE, R. B., KARIG, D. K., MCGREGOR, J. P., AND SHI, Z. Enlisting Hardware Architecture to Thwart Malicious Code Injection. In *Proceedings of the International Conference on Security in Pervasive Computing (SPC-2003), Lecture Notes in Computer Science, Springer Verlag* (March 2003).

[12] LI, T., ELLIS, C. S., LEBECK, A. R., AND SORIN, D. J. Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution. In *Proceedings of the USENIX ATC* (April 2005), pp. 31–44.

[13] LOCASTO, M. E., SIDIROGLOU, S., AND KEROMYTIS, A. D. Application Communities: Using Monoculture for Dependability. In *Proceedings of the* 1st *Workshop on Hot Topics in System Dependability (HotDep-05)* (June 2005).

[14] LOCASTO, M. E., STAVROU, A., CRETU, G. F., KEROMYTIS, A. D., AND STOLFO, S. J. Quantifying Application Behavior Space for Detection and Self-Healing. Tech. Rep. CUCS-017-06, Columbia University, 2006.

[15] NEWSOME, J., AND SONG, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the* 12th *Symposium on Network and Distributed System Security (NDSS)* (February 2005).

[16] NIGHTINGALE, E. B., CHEN, P., AND FLINN, J. Speculative Execution in a Distributed File System. . In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP 2005)* (2005).

[17] OPLINGER, J., AND LAM, M. S. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the* 10th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)* (October 2002).

[18] PATIL, H., AND FISCHER, C. N. Efficient Turn-time Monitoring Using Shadow Processing. In *Proceedings of the* 2nd *International Workshop on Automated and Algorithmic Debugging* (1995).

[19] PROVOS, N. Improving Host Security with System Call Policies. In *Proceedings of the* 12th *USENIX Security Symposium* (August 2003), pp. 207–225.

[20] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP 2005)* (2005).

[21] REYNOLDS, J. C., JUST, J., CLOUGH, L., AND MAGLICH, R. On-Line Intrusion Detection and Attack Prevention Using Diversity, Genrate-and-Test, and Generalization. In *Proceedings of the* 36th *Hawaii International Conference on System Sciences (HICSS)* (2003).

[22] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D., LEU, T., AND W BEEBEE, J. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings* 6th *Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004).

[23] SIDIROGLOU, S., LOCASTO, M. E., BOYD, S. W., AND KEROMYTIS, A. D. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference* (April 2005), pp. 149–161.

[24] SMIRNOV, A., AND CHIUEH, T. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *Proceedings of the* 12th *Symposium on Network and Distributed System Security (NDSS)* (February 2005).

[25] SOMAYAJI, A., AND FORREST, S. Automated Response Using System-Call Delays. In *Proceedings of the* 9th *USENIX Security Symposium* (August 2000).

Notes

[1]This instance of the Halting Problem is typically solved by getting impatient and terminating supervision or the supervised program. Other strategies may include perturbing the environment, as suggested by the Rx system.

[2]Such supervision does, of course, enable a virX to intercept system calls, and a hybrid approach combining both machine-level and system-call supervision is probably most effective.

[3]Given appropriate hardware support, entry and exit could be implemented as assembly language instructions.

[4]`http://www1.cs.columbia.edu/~locasto/research/virtxf/`