Protection is a Software Issue

Brian N. Bershad Stefan Savage Przemysław Pardyak
David Becker Marc Fiuczynski Emin Gün Sirer

{bershad,savage,pardy,becker,mef,egs}@cs.washington.edu

Dept. of Computer Science and Engineering FR-35
University of Washington
Seattle, WA 98195, U.S.A.

1 Some misconceptions about protection

There is a misconception in much of the operating systems community that hardware mechanisms are the only way to ensure system integrity in the presence of malfunctioning or malicious code [Cheriton & Duda 94, Golub et al. 90, Cheriton & Zwaenepoel 83]. For example, for almost 10 years we've heard tales of how microkernels are more reliable than monolithic systems because they rely on hardware implemented address space boundaries to enforce protection between independent subsystems. Despite this, our microkernel-based systems seem to crash about as often as our conventional systems. A lot of our friends say the same thing. Before microkernel systems, layered protection hierarchies were asserted to have greater reliability. Intel x86 processors provide a layered protection model, and yet in practice operating systems using this processor do not exhibit exceptional degrees of robustness. And then, of course, every few years brings another suggestion for a hardware-based capability system together with the always unfulfilled promise of increased system reliability [Berstis 80, Organick 83]. These experiences have revealed that hardware supported protection does not inherently improve the reliability of complicated software systems.

The reality is that modern operating systems are strongly dependent on software mechanisms to protect system resources from users. This is true despite the fact that the promoters of these systems imply that their reliability and integrity derive solely from the use of a core set of protected hardware mechanisms, such as address spaces and protected supervisor mode [Custer 93, Cheriton & Duda 94]. While typical microprocessors provide cheap and effective hardware mechanisms to protect the "load word/store word" interface, operating systems are forced to abstract and virtualize this interface to export a far richer set of resources such as files, sockets, threads, and consoles. The access semantics for these resources are almost always protected by software checks and not hardware. For example, protection issues such as "should I allow this process to terminate that process?", "is there enough memory to spawn a new task", "has that thread had more than its share of CPU time", "can I write to that file" or "can this process really halt the machine?" are not typically resolved automatically in hardware. Processor architectures simply do not provide enough fine-grained control over access to shared system resources to ensure that a program only accesses the resources to which it is allowed.

1.1 Position statement

Our position is that software protection mechanisms are not only necessary, but have inherent advantages over hardware for enforcing the protection requirements of an operating system. Software is flexible, explicit, precise, and in many cases, open to incredible optimizations. By contrast, hardware mechanisms are rigid, implicit, imprecise, and unoptimizable.

To see this, consider the implementation of a typical window system which isolates access to individual rectangular regions of the screen. Hardware protection mechanisms can accomplish this goal, but at significant cost. This cost arises because the hardware mechanism, a protected virtual memory page, is rigid, implicit and imprecise for the task of protecting window relative operations. It is imprecise because it is limited to addressing contiguous address ranges, rigid because it can not be easily altered to address rectangular non-contiguous columns of memory, and implicit because the programmer uses the memory interface and not window operations. It is also hard to optimize and the cost of protection is frequently incurred even when a safe operation is being invoked.

A software interface, like the UNIX X-Windows system, is generally able to provide higher performance because its protection implementation is specialized for window relative operations. In the X server, windows have a simple and inexpensive representation (the integer), and their interface requires programmers to be explicit about the action they require. This al-

lows the server to safely manage screen resources and clip user interactions to the boundaries of a particular part of the screen. Moreover, the cost of this protection may be amortized over the length of an entire operation. For instance, a line drawing operation can eliminate most checks if both end points are found to lie within the window. Hardware mechanisms are appropriate for coarse grain isolation of the entire screen buffer, but finer grain control is far better suited to software. For this reason, the X server is typically implemented by mapping the whole of the frame buffer into its own privileged virtual address space and validating client requests at run time.

2 Different kinds of protection mechanisms

We make the observation that there are basically two kinds of protection mechanisms. Those which implicitly restrict the ability to name a resource and those which explicitly prevent a named resource from being misused. In modern systems these mechanisms are implemented in both hardware and software as follows:

Conditionals – fine-grained access control through explicit "if" statements embedded in the source code.

Data scoping – per-process data structures which implicitly restrict the set of resources a process can access.

Address spaces – coarse-grained support to implicitly control the naming of physical memory.

Memory Protection – coarse-grained support for explicit control of read, write, and execute access to nameable addresses.

Clearly, the first two mechanisms are "software mechanisms" in that they are enforced by the programmer or a compiler. The mechanisms come in the form of additional logic in the operating system that ensures that programs do not misbehave with respect to a set of high level semantics about shared resources.

The last two mechanisms, usually implemented in hardware, ensure that programs do not directly share large regions of memory that are not intended to be shared, but allow them to share memory in limited ways. This functionality may also be provided through software using pointer-safe languages [Nelson 91] or software fault isolation [Wahbe et al. 93] to restrict the set of addresses that may be referenced.

Choosing the protection mechanism for a particular resource depends on both the relative execution cost of protection implemented in hardware or software, as well as the appropriateness to the task. Hardware mechanisms such as address spaces and memory protection are well suited to isolating large regions of contiguous physical memory when changes in protection are relatively infrequent. Fine-grained protection, protection that changes frequently, or protection of resources which are not easily represented through contiguous ranges of memory, are best implemented in software, as it is flexible, has lower execution cost, and is much easier to optimize.

There is a synergy between the protection provided by hardware and software. Software mechanisms usually rely on hardware as a foundation to ensure their own integrity, while changes in hardware protection are usually controlled and limited through software mechanisms. Specifically, any software mechanism which is itself not protected by another software mechanism must be protected by hardware, otherwise it is useless. For example, those important "if" statements in the file system which implement access control facilities would be meaningless if any program could cause them to always return true. Hardware memory protection is likewise useless without software conditionals to control it. For instance, the capability to map the screen buffer into an address space is limited by hardware, but ultimately controlled by those important "if" statements which govern whether or not `mmap(ScreenBuffer)` is allowed for a particular process.

3 Static protection mechanisms

Fundamentally, a protection mechanism exists to enforce a constraint against what a program can do. If that constraint can be statically assumed, then additional protection mechanisms, such as those described in the previous section, are unnecessary, or can be replaced with simpler ones. We commonly see this behavior within the implementation of internal interfaces of an operating system kernel. There, kernel code implicitly assumes that other pieces of the kernel using that code obey system calling conventions, release locks at the right time, do not pass references to stack-allocated or improperly scoped variables, and so on. Because of these assumptions, the kernel implementation can be streamlined to avoid checks which ensure that protection is not being violated. Many checks can also be enforced at compile time. Techniques such as flow-analysis [Chambers & Ungar 90] and dynamic code generation [Keppel et al. 93] can be used to leverage the compiler even further in the elimination of runtime checks. Moreover, compilers of languages that enforce name scoping allow an even larger number of static assertions to be expressed and verified at compile-time.

Static protection mechanisms are made possible by the existence and assertion of a set of assumptions about the behavior of code. If we could assume that code never behaved badly, then no protection mechanisms would be necessary. If we could assume that code might only violate interface semantics, but never invoke a load or store to a memory location outside of its purview, then we could eliminate address protection checks. If we could assume that code would never name a resource to which it was not granted access, then we could eliminate access checks to resources more complicated than simple memory. If we could assume that code had bounded execution time

then we could run arbitrary fragments of user-code in the kernel without fear of "losing" the processor. Assumptions such as these can improve the structure and performance of incrementally constructed and specialized system code [Massalin & Pu 89].

4 Flexible software protection

Several projects are currently seeking to improve the number and quality of the assumptions that can be made of code that interacts with an operating system [Engler et al. 94, Lucco 94, Bershad et al. 94]. The fundamental idea in these projects is that by constraining the actions of user code, that code can interact more closely with untrusting code that would otherwise require a runtime protection mechanism. For example, by limiting the range of addresses to which a code module can load, store, or jump, and denying it access to privileged machine instructions, it becomes possible to install arbitrary user code into the kernel. By enforcing or deriving invariants about data types, and access protocols through safe design and compiler automation, it is additionally possible to allow untrusted user code to access shared machine resources [Savage & Bershad 94]. The advantage of these approaches is that the cost of hardware protection mechanisms (trap, context-switch, system call dispatch, parameter marshaling) can be eliminated from the picture. Of course, this only becomes true because exactly those functions provided by the user/kernel hardware boundary – what has recently been called "the red line" [Cheriton 94] – have been supplanted by software of equivalent functionality, but greater flexibility. This flexibility allows system software to be incrementally specialized to the needs of its applications in ways that would be impossible, or far too expensive, to implement in systems protected by static hardware mechanisms.

All these projects are in effect replacing one set of assumptions – "code is well-behaved" – with another – "we can ensure that code is well-behaved using automatic software mechanisms such as a compiler." Clearly, if we cannot assume the integrity of the automatic mechanisms, then it is not feasible to rely on them. It has been argued that compilers are notoriously buggy, and that it is unwise to assign them the responsibility for system integrity [Cheriton 94]. While there may exist buggy compilers, not all compilers generate bad code. In fact, serious compilers are often required to "work around" more serious bugs that frequently arise in chip implementations [Crothers 94a, Crothers 94b, Hummel 92].

There is nothing inherent about hardware that makes it more bulletproof than software. Because hardware is not a sufficient mechanism to ensure the integrity of the whole system, software mechanisms remain necessary. The operating system should therefore provide a protection infrastructure that permits those software mechanisms to be reliably applied with minimum overhead.

References

[Bershad et al. 94] Bershad, B. N., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S., and Sirer, E. G. SPIN – An Extensible Microkernel for Application-specific Operating System Services. In *Proceedings of the 1994 European SIGOPS Workshop*, September 1994.

[Berstis 80] Berstis, V. Security and Protection of Data in the IBM System/38. In *Proceedings of the 7th Symposium on Computer Architecture*, pages 245–252, May 1980.

[Chambers & Ungar 90] Chambers, C. and Ungar, D. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically Typed Object-Oriented Language Based on Prototypes. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 150–164, June 1990.

[Cheriton & Duda 94] Cheriton, D. R. and Duda, K. J. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the First USENIX Symposium on Operting Systems Design and Implementation (OSDI)*, pages 179–194, November 1994.

[Cheriton & Zwaenepoel 83] Cheriton, D. R. and Zwaenepoel, W. The Distributed V Kernel and its Performance for Diskless Workstations. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 129–140, October 1983.

[Cheriton 94] Cheriton, D. R. Low and High Risk Operating System Architectures. In *Proceedings of the First USENIX Symposium on Operting Systems Design and Implementation (OSDI)*, page 197, November 1994.

[Crothers 94a] Crothers, B. Faulty FPU flubs math in certain equations. *InfoWorld*, 16, November 1994.

[Crothers 94b] Crothers, B. Multithreading gets lost on P100 systems. *InfoWorld*, 16, November 1994.

[Custer 93] Custer, H. *Inside Windows NT.* Microsoft Press, 1993.

[Engler et al. 94] Engler, D., Kaashoek, M. F., and O'Toole, J. The Operating System Kernel as a Secure Programmable Machine. In *Proceedings of the 1994 European SIGOPS Workshop*, September 1994.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the 1990 Summer USENIX Conference*, pages 87–95, June 1990.

[Hummel 92] Hummel, R. L. *Programmer's Technical Reference: The Processor and Coprocessor.* Ziff-Davis Press, 1992.

[Keppel et al. 93] Keppel, D., Eggers, S., and Henry, R. Evaluating Runtime-Compiled, Value-Specific Optimizations. Technical Report UW-CSE-93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[Lucco 94] Lucco, S. High-Performance Microkernel Systems. In *Proceedings of the First USENIX Symposium on Operting Systems Design and Implementation (OSDI)*, page 199, November 1994.

[Massalin & Pu 89] Massalin, H. and Pu, C. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[Nelson 91] Nelson, G., editor. *System Programming in Modula-3*. Prentice Hall, 1991.

[Organick 83] Organick, E. I. *A Programmer's View of the Intel 432 System*. McGraw-Hill, 1983.

[Savage & Bershad 94] Savage, S. and Bershad, B. N. Some Issues in the Design of an Extensible Operating System. In *Proceedings of the First USENIX Symposium on Operting Systems Design and Implementation (OSDI)*, page 196, November 1994.

[Wahbe et al. 93] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, December 1993.