Lessons from VAX/SVS for High-Assurance VM Systems

Steve Lipner | Microsoft
Trent Jaeger | Pennsylvania State University
Mary Ellen Zurko | Cisco Systems

The authors take a look back at VAX/SVS, a high-assurance virtual machine monitor project, extracting its most pertinent lessons on access control, layering, minimization, and verification of assurance.

In May 1990, "A VMM Security Kernel for the VAX Architecture" was lead paper and Best Paper Award winner at the IEEE Symposium on Security and Privacy.[1] "The Auditing Facility for a VMM Security Kernel" was also presented that year,[2] and the year after, two papers on covert channels, "An Analysis of Covert Timing Channels"[3] and "Storage Channels in Disk Arm Optimization"[4] were presented. The team members said the project, VAX/SVS (Secure Virtual System), was a technical milestone in high-assurance operating systems. However, for business reasons, the project was officially cancelled as a product in February 1990.

More than 20 years later, we discuss what we consider the most important results from VAX/SVS, with an eye toward how it can inform high-assurance system creation today.[1]

Background

By 1981, several government-sponsored research projects had attempted to build high-assurance operating systems.[5–7] Some had been canceled while others continued to subsist on government research funding. None had been deployed operationally or made their way into a vendor's commercial product line.

Paul Karger and Steve Lipner were associated with the US Air Force Multics Guardian Project,[5] intended to modify Multics to incorporate a high-assurance security kernel, and with SCOMP (Secure Communications Processor),[7] intended to build a security kernel-based minicomputer communications front-end for the Project Guardian version of Multics, as well as with early and successful security penetration test projects. Thus, they were aware of both the need for high security and the challenges of achieving it, including achieving application compatibility and maintaining performance. Interestingly, they felt that the market for high assurance wasn't a concern, taking an "if we build it, they will come" perspective. Working in Digital Equipment's Corporate Research Group, Karger led an effort to prototype the integration of mandatory security into VAX/VMS. This project was successful enough that Digital Equipment developed an appetite for additional work on security and hired Lipner to lead the effort.

After the 1981 IEEE Symposium on Security and Privacy, Lipner and Karger talked over dinner and brainstormed alternatives for achieving high assurance. Lipner observed that if a project sought to make a high-assurance system compatible with an existing

 Copublished by the IEEE Computer and Reliability Societies

operating system (for Digital Equipment, VAX/VMS), it would always be a release behind the standard product because the product's development wouldn't stop to allow the new high-assurance version to achieve parity. If the high-assurance version went its own way and ignored compatibility, it would have no applications and fail in the marketplace. The solution to this problem was to build a high-assurance virtual machine monitor (VMM) that would layer underneath the standard product; updates to the standard product would run on the high-assurance version at once.

The notion of a high-assurance VMM wasn't new. In the early 1970s, a team at the University of California, Los Angeles, under Gerry Popek prototyped a VMM security kernel for the PDP-11/45, and a team at System Development Corporation (SDC) under Clark Weissman and Marvin Schaeffer built a kernelized version of IBM's VM/370. Neither of these systems became a commercial product; the UCLA project was never intended to do so, and the SDC project lacked commercial vendor support. With optimism born of naivete (or naivete born of optimism), Lipner and Karger took their idea to the management of Digital Equality's research group, and VAX/SVS was born.

As they considered building a high-assurance VMM, the Digital Equality team focused on the Anderson Report's reference monitor requirements,[8] which specified that a system implementing the reference monitor requirements must

- mediate all accesses by subjects to objects,
- protect itself and its databases from attack, and
- be small enough to be subject to analysis and tests to ensure that it's correct.

A VMM appeared to have significant advantages in simplicity and application compatibility. Karger and Lipner believed that a high-assurance system could mimic the well-specified VAX hardware interface with far less mechanism than it could the VAX/VMS APIs.

By 1981, Dijkstra's THE (Technische Hogeschool Eindhoven) Multiprogramming System and MITRE's Venus system had applied layering to build a reliable system.[9] SRI International's PSOS (Provably Secure Operating System) project had proposed (but not implemented) a layered architecture that would support formal verification of the end system,[6] and the Multics Guardian Project had intended to build a layered system before its cancellation. Several projects under Roger Schell (the leader of the Guardian Project) at the Naval Postgraduate School (NPS) had successfully implemented layered security kernel prototypes.

Given their Guardian Project experience, Lipner and Karger believed that a layered implementation would help VAX/SVS achieve quality, reliability, and security. They also believed, referring back to the PSOS project, that layering would facilitate the resulting system's formal verification. Thus, VAX/SVS made an early commitment to layering. The initial design study proposed a layered design based heavily on the NPS work, and that design survived with relatively few changes until the system's eventual cancellation.

> Theories of security come from theories of insecurity.
> —Rick Proto

VAX/SVS Lessons

Building a high-assurance system means addressing how the system's security controls—and the system as a whole—will achieve a particular level or likelihood of correctness or proper functioning. As a high-assurance system, the VAX/SVS project addressed architectural, design, implementation, and other process aspects. It was designed to meet the requirements for class A1, the highest level of assurance defined in the US Department of Defense (DoD)'s *Trusted Computer Systems Evaluation Criteria* (TCSEC) or the "Orange Book." In addition, we find that the Anderson Report's reference monitor concept runs through all our VAX/SVS lessons; security controls are gathered in a single place, which is always invoked, made tamperproof, and small enough to validate. VAX/SVS required security policy, and targeting a multilevel operating system meant using Bell-LaPadula as its core policy.[10] Other aspects of assurance the Orange Book covered included auditing, design process, testing, documentation, and operational concerns. The highest level of Orange Book assurance required formal methods to be applied to covert channel analysis, design, and test plans as well as trusted distribution (which the team fondly called "trusted trucks"). A comprehensive approach to assurance includes the entity responsible for evaluating the assurance, which in the case of the DoD's Orange Book, was an appropriately accredited third party.

In this article, we concentrate on four lessons from VAX/SVS development: access control, layering, minimization, and verification of assurance.

Verifiable and Tamperproof Access Control

A major consideration of many systems' security controls is access control. At the time of VAX/SVS, nearly all commercial operating systems provided only *discretionary access control*, wherein an object's access rights were determined by the object's owner. It was well known that some security problems couldn't be solved

using discretionary access control. For example, a Trojan Horse in application software could leak an object to unauthorized users by changing the access rights to that object or writing a copy of the object to another object accessible to unauthorized users.

An alternative approach to access control, *mandatory access control*, lets a system administrator constrain an object's access rights. A major goal for high-assurance systems design is to develop an approach by which lattice security models (such as Bell-LaPadula[10]) could be verifiably enforced, implementing the reference monitor concept. By the late 1980s, no commercial operating system strictly met these requirements. Although several high-assurance system prototypes that targeted the reference monitor concept were built in the 1970s and 1980s, poor design choices led to large sizes or poor performance. For example, the KSOS (Kernelized Secure Operating System) kernel and KSOS Unix emulator were each larger than contemporary Unix systems.

The VAX/SVS project aimed to overcome these prior limitations by integrating lattice security model enforcement into a VMM security kernel. Unlike prior attempts to build VMM security kernels, the VAX/SVS system was designed "from scratch," which significantly facilitated the reference monitor concept's implementation.

Key to developing a simple, clean design for the VAX/SVS access control system was choosing a small number of subject and object types. VAX/SVS provided only two types of subjects: users and virtual machines (VMs). Users accessed the security kernel via a trusted path mechanism, and the security kernel performed operations on users' behalf given their particular access rights (as we discuss later). VAX/SVS provided only four types of objects: devices, volumes, virtualized resources, and security kernel files. It exported dedicated volumes and virtual disk volumes to VMs and had its own files on which it controlled access, so only privileged processes could use them.

Subjects and objects were both assigned access classes consisting of a secrecy class and an integrity class. This approach combines the work of Bell and LaPadula for secrecy[10] and Kenneth J. Biba[11] for integrity to prevent information flows that might leak objects to unauthorized subjects or allow modification by unauthorized subjects, respectively. The requirements for A1 evaluation were closely tied to the Bell-LaPadula model for mandatory access control: lower cleared users (or processes) were forbidden to gain access to higher classification information, either by direct access or by exploiting covert channels.

The decision to include the Biba integrity model in VAX/SVS was driven more by theoretical interest than real need. Lipner had published a paper on applying the Biba model to commercial data security problems, so it seemed that there might be real-world requirements. And the cost of incorporating mandatory integrity controls in a system that implemented the Bell-LaPadula model was minimal because both models require similar lattice access control mechanisms. In theory, common programs and read-only databases would be created at high integrity and thus protected from modification, but in reality, VAX/SVS didn't use the Biba model to protect its own code and databases, and we aren't aware that any of the system's "beta test" users applied the controls that enforced the Biba model.

In addition to access classes, subjects in VAX/SVS could also be given special *privileges*. Such privileges let system users (for instance, administrators) perform security-critical actions, such as managing the assignment of access classes and allowing security policy modification.

Using the Bell-LaPadula model, VAX/SVS provided single-level VMs. If users needed to process at a higher level while reading lower-level information, they would connect to a higher-level VM, which could attach a lower-level virtual disk as a read-only device.

The VAX/SVS security kernel is the reference monitor that enforces this lattice policy model (extended with privileges), so a key question is how well did the VAX/SVS achieve the aims of the reference monitor concept? Complete mediation benefits from the small, fixed number of object types in the VMM security kernel, making it much easier to ensure that all the relevant security-sensitive operations are mediated. A challenge that resulted from the choice of a small, simple VMM system was that the granularity of object access control and sharing was the VM and virtual disk drive. The VAX/SVS development team felt it had plausible approaches to implementing a usable system despite the coarse object granularity.

The VAX/SVS security kernel provided tamperproof execution by cleanly separating the security kernel and the subjects (users and VMs). All the trusted code in the VAX/SVS system ran in the security kernel, so the designers could focus on protecting the entry points. The only trusted entities in the VAX/SVS system were devices and subjects with privileges. Devices were privileged if they had the ability to perform direct memory access (DMA), which permitted them to write to any physical memory address. As a result, devices and their drivers were part of the trusted computing base, so the VAX/SVS team had to develop and maintain the device drivers. Subjects with privileges could modify the lattice security policy and other configurations, which obviously had profound implications. To prevent vulnerabilities, privileges with potential impact on a system's security policy were only accessible via a trusted path, so the security kernel could authenticate that a real user

was behind the action. Two privileges were reserved for VMs, so these VMs needed to be privileged. The VAX/SVS designers were primarily concerned with ensuring that mandatory security could be enforced. Because performing these two privileged operations outside the kernel didn't violate that goal, the designers preferred limiting these abilities rather than creating a larger and more complex kernel.

In addition to mandatory access control and user privileges, VAX/SVS implemented *discretionary access control* (access control lists) on objects. The discretionary access control's effectiveness was the only major area of disagreement between the development team and TCSEC evaluators. The design was optimized around *multiuser VMs*—if each user required his or her own VM, the physical memory requirement for a commercially viable VAX/SVS system would have grown beyond what was available, and the team would have had to modify the design to demand page VMs' memories. In the chosen design, a VM would operate at a specific mandatory access class (level and category set) with read-write access to objects at that access class and read access to objects at lower confidentiality access classes (that is, lower security level and lesser category set). All users who needed to access objects at that VM's access class would share the machine.

With multiuser VMs, VAX/SVS couldn't determine with high assurance which individual user took a specific action for the purpose of enforcing discretionary access control or collecting an audit trail. The VAX/SVS team argued that, given the reality of Trojan horses, the security gain wasn't worth the impact on the system. The evaluators argued that the TCSEC required "A1 discretionary access controls." In the end, the team documented a way to configure a VAX/SVS system for single-user VMs, with the expectation that user organizations would configure their VAX/SVS systems for more efficient and adequately secure multiuser VMs. We note that the tradition of documenting an evaluated configuration with specific security attributes, knowing full well it will be largely impractical, continues today.

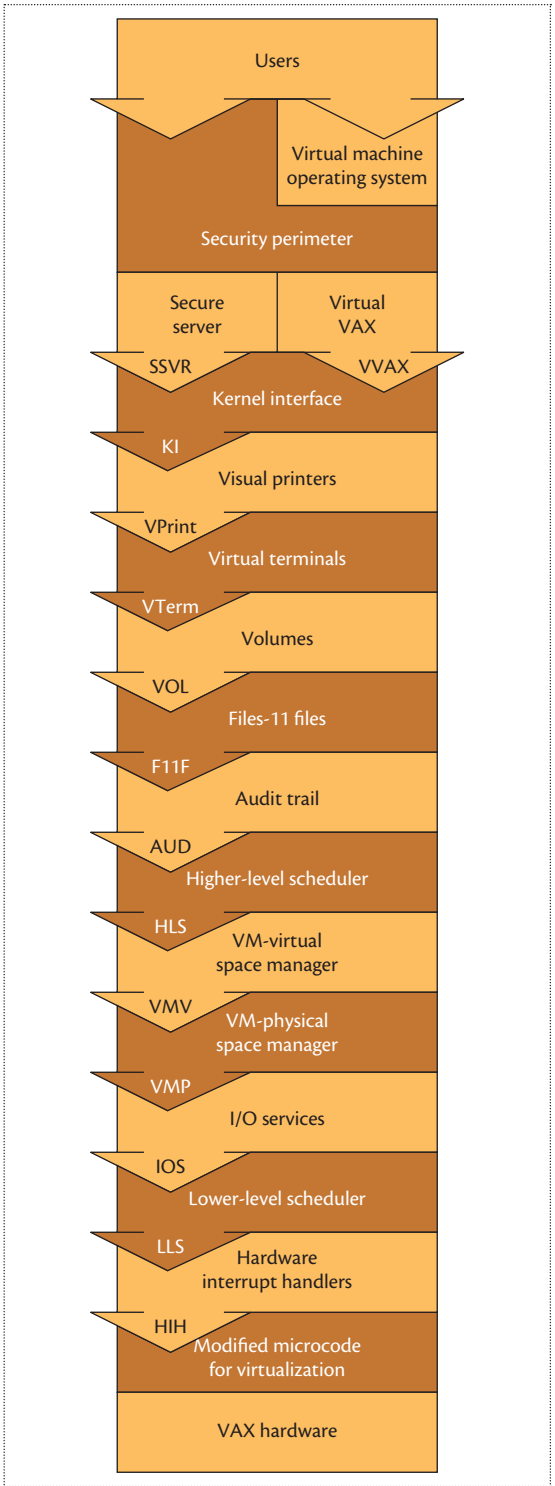

Figure 1. VAX/SVS layers (from "A VMM Security Kernel for the VAX Architecture"[1]). Each layer implemented a well-defined abstraction in the system. Higher layers could call on the services of lower layers whereas lower layers were forbidden from calling on higher layers.

Layered Design

The VAX/SVS layered design approach proved to be key to a number of areas. The Orange Book called for significant use of layering, abstraction, and data hiding. A levels-of-abstraction approach in security kernel design was recommended as a means to reduce complexity and an aid in precise and understandable specifications. Reduced complexity was a core principle of VAX/SVS; the team lived by the "keep it simple, stupid" motto. The team followed other classic layered design principles, including a separation of concerns between the layers, low coupling between layers and high cohesion within them, and limited exposure to layer internals.

Each level of abstraction was a layer that could call any of the lower layers; lower layers were never allowed to call higher layers. The total number of (potential) interactions in the system was conceptually bounded and restricted; because each layer defined its external API, a layer could call a lower layer only through one of the defined entry points. For performance reasons, the team didn't enforce making calls through intervening layers that merely passed the call on to a yet lower layer, though we did briefly consider it. Today, discussions of layered design tend to introduce additional complexity by allowing a rich tree of objects. VAX/SVS was an almost pure sequence of single layers (see Figure 1). The absolute simplicity of the system's layering gave the VAX/SVS design some of its power as a structure for both call flow and overall system organization. A layer's cohesion was based on functionality that naturally needed a great deal of shared code or concepts.

Another benefit of layering is the ability to more easily test a layer in isolation, because the entry points are well defined and, at most, a test environment need only stub out all lower layer entry points. Layering dampens the overall effect of a system's code changes and enhances the interfaces' stability. The grouping and structure that aids understandability could therefore help maintainability (a potential benefit the team didn't get a chance to see). Reuse—another potential benefit of layering—wasn't a concern; the layers were designed for the single system.

Testing emphasized layer entry points—what they would do and what assumptions they made. The layered design defined the API's assumptions and explicitly checked those assumptions "first thing" if they were security relevant. This meant each layer's interface design required an understanding of both functionality and security requirements. The defensive posture at each layer created classic "defense in depth" at specific points in the architecture and call paths. The developers' test environment was such that a layering violation would cause a basic smoke test for a new build of the system to fail. The team agonized over ways to make a rigorously layered system perform well, but they stuck to the layering paradigm.

> [A] mathematical model of the growing embryo will be described. This model will be a simplification and an idealization, and consequently a falsification. It is to be hoped that the features retained for discussion are those of greatest importance in the present state of knowledge.
>
> —Alan Turing

The full system structure and rigor imposed by the choice of a layered design and its subsequent benefits can be contrasted with the Agile approach to software development, which is currently getting a good deal of attention.[12] Although Agile's emphasis on test-driven design is consonant with the testability of each layer's boundaries, there seems to be no aspect of Agile that addresses overall system structure and simplicity. VAX/SVS's layered design let the team share a common understanding of the code base's structure, functionality, and goals, providing a foundation for the discussions on code placement and call paths.

Conway's law states, "Organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations."[13] This indicates that a software system's interface structure will reflect the social structure of the organization that produced it. In VAX/SVS, each layer had an owner, and design conversations often involved layer owners. Team members were partitioned throughout the layers and would move between them as the functionality built up over time. As Conway's law suggests, the system layers reflected the team members' groups. Potential stakeholders and reviewers for design and code changes were at the layers above the one changing (or team members at the changing layer if the change didn't affect the interface)—another way our communication structure reflected our system design. This seemed unremarkable at the time. Conversely, if Conway's law is true, the communication structure of a system produced today through the Agile methodology is likely to be diverse and unstructured, and thus more complex and harder to understand, reason about, and predict.

The counterarguments to layered design include lack of engineering flexibility, performance impact, the difficult requirement of defining the layers up front, time to market, and the ability to make rapid changes. In their limited experience of a version 0 project, the VAX/SVS team didn't experience any severe constraints on engineering flexibility. In this article, we touch on performance and time to market in general.

Minimization

Of the three reference monitor requirements, the requirement that the system be small enough to be subject to analysis and tests to ensure that it's correct is probably the most challenging for secure system developers. VAX/VMS purported to mediate every access by a subject (process or user) to an object (file, device, or interprocess communication [IPC] channel) but its size and complexity were great enough that the presence of exploitable vulnerabilities was a certainty. The choice of a VMM architecture was intended to support minimization of the trusted code base.

However, choosing to build a VMM was only the beginning of the quest for minimization. From the development project's beginning, the team intended to minimize the amount of trusted mechanism in the system. Three examples—memory management, I/O, and user interface—illustrate this point. The first two date back to the 1981 design study that set the overall direction of the project, and the third was a result of the quest to provide the system with a degree of usability.

Memory management. Karger, Lipner, and Andrew Mason were three key participants in VAX/SVS's initial design study, and all were veterans of the Multics Guardian Project. Multics required both segmentation and demand paging to provide each process with a rich application environment while operating on hardware with limited physical memory. The planned Guardian Project architecture included kernel support for demand paging with fully trusted kernel processes (virtual processors) moving page frames between disk and main memory. The resulting system would have been relatively complex.

The Guardian Project veterans' initial memory management design for VAX/SVS included support for demand paging of VMs' physical memory spaces, similar to IBM's VM/370. Peter Conklin, one of the original VAX/VMS architects, participated as a guest in the initial design study. When he saw the plan to include paging in the VAX/SVS kernel, he pointed out that VAX/VMS (which would be running in each VM) implemented paging and that having two independent paging systems was likely to result in poor performance. A VAX/SVS design that did away with paging would be much simpler. He also observed that providing communication between VAX/VMS and VAX/SVS to optimize performance would result in an even more complicated kernel design. Finally, he pointed out that physical memories were getting larger as hardware costs dropped and that it would eventually be feasible to just give each VM a static allocation of physical memory.

The VAX/SVS design adopted Conklin's recommendation, which was referred to internally as "memory is cheap." As a result, the VM physical memory management and virtual memory management layers were significantly simplified. As Conklin predicted, physical memory sizes continued to increase and prices to decrease during the life of the project, and the project team never regretted the decision to abandon demand paging in the kernel.

Input/Output. From the earliest days of VMMs (IBM's CP-67), I/O management has presented challenging problems. I/O is a "sensitive" function on any VMM system, so the VMM must be able to intercept and interpret each I/O operation. Whereas IBM mainframes implement discrete privileged I/O instructions (that can be made to trap to the VMM), Digital Equipment computers' (PDP-11s and VAXs) I/O is controlled by using ordinary (unprivileged) instructions to read and write specific physical addresses that correspond to I/O control registers rather than memory. Thus, the VMM developer for a Digital Equipment computer needed to intercept every read or write to an I/O register location and interpret the intended operation. The resulting code is both slow and complex.

VAX/SVS's initial design anticipated virtualizing I/O operations that VAX/VMS would direct toward standard VAX devices. Karger sketched an adapter that would map VMs' I/O operations and minimize software intervention. When Conklin saw that design and considered the problem of I/O from a VM, his reaction was, "Don't do that—just create a special call from the virtual machine to the VMM that will request an I/O operation and provide the necessary parameters. The VMM can interpret the request in one operation and it will be much more efficient." In today's jargon of VMs, this would be referred to as "enlightening" the VM's operating system to rely explicitly on the VMM for I/O.

Because VAX/VMS was designed to be highly adaptable to new kinds of I/O devices, Conklin's suggestion was both feasible and easy to implement. The resulting I/O architecture was vastly simpler and performed better than an alternative that would have required interpretation of individual I/O register operations. When Ultrix (Digital Equipment's version of Unix) was eventually ported to VAX/SVS, the wisdom of this design choice was demonstrated again—the entire port required only a few weeks' effort by a small subset of the Ultrix development team.

User interface. As the VAX/SVS team made the transition from demonstrating that it was possible for VAX/VMS to run in a prototype VMM to building a usable A1 system, they realized that a large number of system administration operations would be required and that the path from administrator to system state would need to be trusted. It seemed likely that the resulting command parsers would be large and complex: command parsing might require as much code as the rest of the security kernel, even with a relatively primitive user interface (the technology of the day was command line rather than GUI).

After some discussion, the team came up with the idea of parsing administrator commands in an untrusted application running on a VM, and having that application pass the parsed commands to the kernel in a simple and standardized format. The kernel would then display the command to the administrator for confirmation,

and the only administrator command the kernel would have to parse would be a "confirm" or "cancel." A secure server attention key and associated protocol allowed the administrator to be certain that he or she was interacting with the kernel rather than an untrusted program spoofing the kernel.

The VAX/SVS team was aware that the system needed to be usable, so the actual user interface included kernel parsing (without need for confirmation) of commands that ordinary users would use frequently, such as "connect me to another VM" and "logout." The system presented ordinary users with a very natural interface—as though they were using a terminal concentrator and switching from a machine at one security level to one at another.

Verification of Assurance

Only one or two systems completed evaluation at TCSEC class A1, providing few success stories to emulate. VAX/SVS's design was extremely simple, even for its day. The team was inculcated with the importance of adhering to the Bell-LaPadula model and a layered system architecture, both seen as key to verification. Layering played an important part in the required system specifications. The formal model required a descriptive top-level specification (DTLS), a complete natural-language description of the system. The per-layer design (and API) descriptions formed a substantial part of the required DTLS.

Originally, the VAX/SVS team anticipated that the layered design would support formal verification. The assumption, based on the concepts articulated in the PSOS research,[6] was that each layer would be verified to correctly implement a specification using code in its own layer that invoked services provided by lower layers. Verification of each layer would lead to verification of the entire system. Unfortunately, verification technology of the 1980s didn't follow up on the promises of the 1970s, and no formal verification tool available to the VAX/SVS team was capable of taking advantage of the system's layered structure. As a result, formal verification was confined to analysis of the system's external interfaces against the Bell-LaPadula model requirements. This level of formal verification was typical for high-assurance systems of the era.

The VAX/SVS security kernel was designed and implemented with the goal of comprehensive verification. Extensive testing was deployed for regression and various use cases. The system was formally specified as part of the A1 assurance process using InaJo. The VAX/SVS security kernel implementation consisted of approximately 48,000 source lines of code (SLOC), which were written in PL/1, Pascal, and Macro-32 assembly language. Because there were more than 11,000 source lines of assembly code, evaluating assurance was a significant undertaking. The recent formal assurance of the seL4 system shows that the expense per line of code is still high for formal assurance (9,300 SLOC at US$10,000 per LOC).[14] The seL4 system is a microkernel rather than a VMM, so it provides less functionality.

The formal assurance process was further complicated by the need to evaluate new code when a new device was added to the system. The availability of device support has been a critical factor for the adoption of operating systems over the years, so this would have been a major challenge for maintaining assurance of the VAX/SVS security kernel. Now, the introduction of IOMMU (input/output memory management unit) hardware to commercial processors means that device drivers and their inherent challenges (for example, controlling DMA) can be moved to the user space (even to unprivileged VMs). However, if a device is needed by trusted code, it will still need to be in the trusted computing base.

The TCSEC required that an A1 system limit covert channels' bandwidth, so the team undertook a significant effort to eliminate all covert storage channels and mitigate covert timing channels in VAX/SVS.[3] The VAX/SVS interface's simplicity and the team's strict attention to adhering to the Bell-LaPadula model resulted in a system that was relatively free from classic storage channels. (Many of VAX/SVS's resource allocation mechanisms were either static or implemented by human administrators—a set of choices that aided both simplicity and freedom from storage channels.) Whereas storage channels were only a limited problem, timing channels proved to be a source of surprises, frustration, performance challenges, and project delays. The formal verification work, led by consultant Richard Kemmerer of the University of California at Santa Barbara, applied the Shared Resource Matrix method of identifying covert channels, giving the team a rough sense of what channels might be present. The team collaborated with Robert Morris (of the US National Security Agency) to identify an approach to mitigating timing channels—called "fuzzing clocks" or "fuzzy time." Implementing this approach required significant changes late in the development cycle, which degraded the system's (already marginal) performance. What's worse is that a year after the team published the "fuzzy time" approach, a researcher published an approach to defeating it.[15]

Although the VAX/SVS team met the formal verification requirements for TCSEC class A1, they believed that the actual assurance came from adherence to the layered design principle, thorough documentation, and careful coding. Designs were documented before they

were implemented (unlike the practice at lower levels of the TCSEC or Common Criteria). The team reviewed all the security kernel design decisions and code at all stages of the project and before it was checked in. The coding languages for the system precluded buffer overruns, and the style guides to which the team adhered constrained implementation to conservative and safe practices.

Cancellation

The VAX/SVS development project was intended to produce a commercially viable system that could complete evaluation at TCSEC class A1 and be sold to customers in sufficient quantity to recover its development costs. By 1989, the system was on track to complete evaluation and sufficiently polished to enter a field (beta) test with customers. The field test was reasonably successful: customers were able to use the system, and there was even a rumor that one of the test customers had deployed VAX/SVS in a "multilevel secure" operational configuration.

Despite this level of accomplishment, VAX/SVS was canceled because the business case for the system wasn't sufficient.[16] Even though a great deal of money was spent bringing the system to the point at which customers could use it, sales projections weren't encouraging. Some customers were willing to buy copies of the system, but neither the number of customers nor the number of copies was sufficient to make a profitable business case. At the time, US export restrictions on high-assurance products received much of the blame for cancellation, but the reality was that US and other customers who were eligible to buy VAX/SVS weren't all that interested. Had a decision been made to release the system commercially, that decision would have implied a commitment to maintain and enhance it over period of years. With inadequate sales, the system would've continuously lost money.

To understand why customers didn't want to buy VAX/SVS, we must consider the time in which the system would have come to market. In the late 1980s, customers were beginning to demand personal computers or workstations, networking, and GUIs. VAX/SVS was designed as an isolated time-sharing system that supported users at alphanumeric terminals. A "hack" allowed networking of individual virtual machines through dedicated asynchronous terminal lines, but the system itself wasn't networked.

Modifying VAX/SVS to support workstations, networks, or GUIs would've been a significant development task. Although workstation support would have been the simplest task, it would have required developing and manufacturing a VAX microprocessor with the SVS-specific virtualization features. And workstation support wouldn't have been viable without adding GUI support. The experience of building and evaluating VAX/SVS made it clear that adding networking and GUI support would have required significant research projects—the team would have had to develop concepts, implement them, and "sell" them to the evaluators. It seemed probable that the process would have taken so long that by the time the features could be shipped, user expectations would have moved beyond what VAX/SVS could provide. Of course, this was the very trap that the VMM approach was intended to avoid, but in the end, it seemed unavoidable. We leave it to the reader to judge whether this trap is a fundamental flaw of high-assurance systems.

> The dependency graph, which Dave Parnas called the "Uses" hierarchy, becomes a powerful tool for evaluating the robustness of a system. Module A depends on module B if the correctness of A requires that B performs correctly. In verification terms, B becomes a lemma for the theorem that A performs correctly. A dependency graph with circularities reduces system robustness, just as circularities in reasoning produce politicians.
>
> —Earl Boebert

Should We Build Virtualization Kernels Today?

VAX/SVS served two roles: host security kernel and virtualization manager. One question we should consider in hindsight is whether requiring both roles in a security kernel is practical. The VAX/SVS design resulted in a code base of less than 50 KLOC. Modern, fully functional virtualization kernels are much larger still. For example, the Xen hypervisor had approximately 300 KLOC in 2008. Given the greater size and functionality of modern virtualization kernels and the current cost and complexity of formal assurance, it seems unlikely that a fully assured security kernel with fully functional virtualization could be built today.

Recent hardware advances in security and virtualization might significantly aid the task of separating kernel and virtualization functionality. By adding virtualization support, hardware architects ensure that all sensitive instructions are now privileged, removing the need for I/O emulation. Finally, and perhaps most important, IOMMUs enable DMA devices to be securely removed from the trusted computing base. With the broadly available support for trusted computing mechanisms, it's now possible to measure each software layer independently, letting remote parties verify the system boot process consisting of multiple layers.

> Perfection is finally attained, not when there is no longer anything to add, but when there is no longer anything to be taken away.
>
> —Saint Exupery

So, do such advances make it practical to separate the kernel and virtualization functionalities into two distinct software layers? In the 1990s, second-generation microkernel designs, such as L4 (predecessor of seL4) and Exokernel, focused explicitly on minimizing kernel code. In such kernels, physical resources were partitioned among isolated domains that could communicate through fast IPC primitives. Researchers found such systems effective for constructing optimized mechanisms for hardware use, particularly for network devices. On the other hand, deploying general-purpose systems on such kernels introduced additional performance overhead and development complexity that didn't seem to warrant the benefits, particularly because these kernels were still prone to DMA attacks. However, hosted operating environments, such as L4Linux, showed that the performance overhead of systems with a single physical resource manager was modest (less than 10 percent in 1997). Further hardware advances have ameliorated the effects of these performance costs, reduced vulnerability to DMA attacks, and made it easier to layer software. As a result, it's still an open question as to when security kernel and virtualization functionality should be combined into a single VMM security kernel.

How Do We Obtain Systemwide Access Control?

To implement the reference monitor concept in the security kernel, VAX/SVS provides a rich access control model enforced by a reference validation mechanism. Modern systems aren't designed with a rich access control model at inception, but instead access control is incrementally added to systems as they mature (and adversaries show developers where authorization is necessary). Is it possible to add a reference validation mechanism later in the system development life cycle and still achieve the reference monitor concept? The emergence of program analysis for security might help answer this question.

Once a proper reference validation mechanism is in place, a challenge is managing privilege delegation from the security kernel up to VMs. In the VAX/SVS design, the team tried to limit the trust in user-space code, but modern systems often have a privileged VM, which is tantamount to a complete operating environment running with privilege outside the VMM. Although we have mechanisms to enforce security decisions in privileged VMs, we don't know which software is capable and worthy of being trusted with those decisions. Furthermore, this software is far too complex for formal assurance. Finally, although reference validation mechanisms are being added to a variety of software in VMMs, operating systems, middleware, and applications, these individual access control mechanisms aren't yet integrated into a systemwide mechanism for managing privilege.

How Do We Assure System Security?

Fundamental to the VAX/SVS development process was the task of formal assurance. Unfortunately, this task has not evolved much since the early 1990s. A great deal of manual effort is necessary to convert a system into a format suitable for assurance. In practice, systems can't be formally assured unless they're built with that assurance from the outset. The result is a slow, laborious process whose results might be obsolete on delivery.

In developing secure systems, not all software is equal, and that might make a difference. Gernot Heiser of the seL4 project conjectured that much of the user-space software would be efficient if it was developed in the language used for formal analysis in the seL4 evaluation (Haskell), then compiled into C.[14] However, the performance-critical software, such as the microkernel, would have to be handcrafted code. The implication is that labor-intensive formal assurance might only be necessary to a small subset of performance-critical code; the remaining code could be developed using compilation tools that would check for security properties. This optimistic view misses at least two key limitations. First, someone still has to articulate the security properties that must be achieved for the software and the data that it processes. Second, people tend to prefer programming in languages that are less constrained, but this leads to more security problems. Researchers have long advocated using more structured programming languages to improve code security, or at least enable automated verification, without success. Some recent research focuses on making low-level languages amenable to various security analyses, such as C Intermediate Language (CIL) and Low-Level Virtual Machine (LLVM). Ideally, such techniques will further extend to improve our ability to limit the amount of code that requires manual formal verification.

We find several lessons from VAX/SVS are worth emphasizing and sharing with the broader community today. The reference monitor concept from the Anderson Report provides useful architectural principles for high-assurance systems. Verifiable and tamperproof access control was and remains challenging, in part

because of the diversity of operational requirements. Layering provides many critical benefits that seem to be otherwise lost, such as reducing the number of entry points to defend and test. Minimization requires whole system thinking as well as accurate powers of prognostication. Verification of assurance remains a complex and multifaceted challenge. And business realities interact with all of these considerations: if it takes so long to build a highly assured system that it is no longer competitive with less assured alternatives, customers will refuse to buy or use it. We believe there are many useful lessons from the VAX/SVS work, and we hope and expect they will inform future successful efforts. ■

Acknowledgments

Each of the authors has fond memories of Paul Karger. Without him, this article would not have been written. Peter Conklin, whose contributions to VAX/VMS from Star to Alpha are relatively well known, was an unsung hero of the SVS project. His early insights had a tremendous impact on the system.

References

1. P.A. Karger et al., "A VMM Security Kernel for the VAX Architecture," *Proc. IEEE Symp. Research in Security and Privacy*, IEEE CS, 1990, pp. 2–19.
2. K.F. Seiden and J.P. Melanson, "The Auditing Facility for a VMM Security Kernel," *IEEE Symp. Research in Security and Privacy*, IEEE CS, 1990, pp. 262–277.
3. J.C. Wray, "An Analysis of Covert Timing Channels," *Proc. IEEE Symp. Security and Privacy*, IEEE CS, 1991, pp. 2–7.
4. P.A. Karger and J.C. Wray, "Storage Channels in Disk Arm Optimization," *Proc. IEEE Symp. Security and Privacy*, IEEE CS, 1991, pp. 52–61.
5. N. Adleman et al., *Security Kernel Evaluation for Multics and Secure Multics Design, Development and Certification, Semi-annual Progress Rept. 1 Jan-30 June 76*, report NTIS AD-A038 261/4, Honeywell Information Systems, Aug. 1976.
6. R.J. Feiertag and P.G. Neumann, "The Foundations of a Provably Secure Operating System (PSOS)," *Proc. Nat'l Computer Conf.*, AFIPS, 1979, pp. 329–334.
7. L.J. Fraim, "Scomp: A Solution to the Multilevel Security Problem," *Computer*, vol. 16, no. 7, 1983, pp. 26–34.
8. J.P. Anderson, *Computer Security Technology Planning Study*, report ESD-TR-73-51, MITRE, Air Force Electronic Systems Division, Hanscom, 1972.
9. E.W. Dijkstra, "The Structure of the 'THE'-Multiprogramming System," *Comm. ACM*, vol. 11, no. 5, 1968, pp. 341–346.
10. D.E. Bell and L.J. LaPadula, *Secure Computer System: {Unified} Exposition and {Multics} Interpretation*, report ESD-TR-75-306, Deputy for Command and Management Systems, HQ Electronic Systems Division, Mar. 1976.
11. K.J. Biba, *Integrity Considerations for Secure Computer Systems*, report ESD-TR-76-372, MITRE, Apr. 1977.
12. C. Larman, *Agile and Iterative Development: A Manager's Guide*, Addison-Wesley, 2004, p. 27.
13. M.E. Conway, "How Do Committees Invent?" *Datamation*, vol. 14, no. 5, 1968, pp. 28–31.
14. G. Klein et al., "seL4: Formal Verification of an OS Kernel," *Symp. Operating Systems Principles*, ACM, 2009, pp. 207–220.
15. I.S. Moskowitz and A.R. Miller, "The Influence of Delay upon an Idealized Channel's Bandwidth," *Proc. IEEE Symp. Research in Security and Privacy*, IEEE CS, 1992, pp. 62–67.
16. P.A. Karger et al., "A Retrospective on the VAX VMM Security Kernel," *IEEE Trans. Software Engineering*, vol. 17, no. 11, 1991, pp. 1147–1165.

Steve Lipner is partner director of program management in trustworthy computing security at Microsoft. He's responsible for Microsoft's Security Development Lifecycle, which is used to achieve security assurance of Microsoft products, and government security evaluations of Microsoft products. Lipner has an SM from the Massachusetts Institute of Technology. He coauthored The Security Development Lifecycle (Microsoft Press, 2006) with Michael Howard. Contact him at slipner@microsoft.com.

Trent Jaeger is an associate professor in Pennsylvania State University's Computer Science and Engineering Department and codirector of the Systems and Internet Infrastructure Security (SIIS) Lab. His research interests include operating systems security and the application of programming language techniques to security. Jaeger has a PhD in computer science and engineering from the University of Michigan, Ann Arbor. He's the author of the book, Operating Systems Security. Contact him at tjaeger@cse.psu.edu.

Mary Ellen Zurko is a security architect and strategist at Cisco Systems. Her research interests include security standards and usable security. Zurko has an MS in computer science from the Massachusetts Institute of Technology. She is chair of the IW3C2, the steering committee for the International WWW Conference series, and is an active steering committee member and organizer of New Security Paradigms Workshop and the Symposium on Usable Privacy and Security. Contact her at mez@alum.mit.edu.

Obesity kills, even in the virtual domain.

—Earl Boebert