Lessons from VAX/SVS for High Assurance VM Systems

Steve Lipner slipner@microsoft.com

Trent Jaeger tjaeger@cse.psu.edu

Mary Ellen Zurko mzurko@us.ibm.com

Abstract

VAX/SVS was a high assurance virtual machine monitor (VMM) project, documented in several published papers from the 1990's. We take a look back, extracting the most pertinent lessons from that work for today. These lessons cover reference monitor architectural principles, approaches to verifiable and tamperproof access control, the benefits of layering, the impacts of minimization and verification, and the business reasons behind its cancellation as a product.

Keywords: Security kernels, Verification, Assurance

Introduction

In May of 1990, "A VMM Security Kernel for the VAX Architecture" [1] was lead paper at the IEEE Symposium on Security and Privacy, and was awarded Best Paper. "The Auditing Facility for a VMM Security Kernel" [2] was also presented that year, 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 project the team members had called VAX/SVS (for Secure Virtual System) represented a technical milestone in high assurance operating systems in a number of ways. However, the project was officially cancelled as a product in February of 1990, for business reasons.

More than 20 years later, we are taking this opportunity to highlight what we consider to be the most important results from [1], with an eye toward how they can inform how high assurance systems are considered and built today.

Background

By 1981, several government-sponsored research projects had attempted to build high-assurance operating systems [5, 6, 7]. Some had been cancelled while others continued to subsist on government research funding. None had been deployed operationally or made its way into a vendor's commercial product line.

Paul Karger and author Lipner had been associated with the Multics Guardian [5] and SCOMP [7] projects as well as early and successful security penetration test projects. Thus they were aware of both the need for high security and the challenges of achieving it. Some of the challenges they found of particular concern were actually achieving a high level of security, application compatibility, and performance. Interestingly, they felt that the market for high assurance was not a concern, taking an "if we build it, they will come" perspective. Working in Digital Equipment's Corporate Research Group, Karger had led an effort to prototype the integration of mandatory security into VAX/VMS. This project was successful enough that Digital developed an appetite for additional work on security and hired Lipner to lead the effort.

After the 1981 IEEE Symposium on Security and Privacy in Oakland, 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 operating system (for Digital, VAX/VMS), it would always be a release behind the standard product since development of the standard product would not 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 was not new. A team at UCLA under Gerry Popek prototyped a "VMM Security Kernel" for the PDP-11/45 in the early 1970s, and a team at System Development Corporation (SDC) under Clark Weissman and Marvin Schaeffer had built a "kernelized" version of IBM's VM/370. Neither of these systems became a commercial product – but the UCLA project was never intended to do so and the SDC project lacked commercial vendor support. With optimism born of naïveté (or naïveté born of optimism), Lipner and Karger took their idea to the management of Digital's research group and VAX/SVS was born.

As they considered the idea of building a high-assurance VMM, the Digital team focused on the reference monitor requirements articulated in the Anderson Report [8]. The system that implements the reference monitor requirements must:

- Mediate all accesses by subjects to objects.
- Protect itself and its databases from attack.
- Be small enough to be subject to analysis and tests to assure that it is correct.

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

A second fundamental choice of the VAX/SVS project was the commitment to a layered implementation. By 1981, Dijkstra's THE Multiprogramming System [9] and MITRE's Venus system had applied layering as a way to build a reliable system. The PSOS project at SRI [6] had proposed (but not implemented) a layered architecture that would support formal verification of the end system, and the USAF Multics Guardian project had intended to build a layered system before its cancellation. Several projects under Roger Schell (the leader of Project Guardian) at the Naval Postgraduate School (NPS) had successfully implemented layered security kernel prototypes.

Given their Project Guardian experience, Lipner and Karger believed that a layered implementation would help VAX/SVS to achieve quality, reliability, and security. They also believed, referring back to the PSOS project, that layering would facilitate formal verification of the resulting system. 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 eventual cancellation of the system.

VAX/SVS Lessons

Building a high assurance system means addressing how the security controls in the system, and the system as a whole, will achieve a particular level or likelihood of correctness or right functioning. As a high assurance system, the VAX/SVS project addressed architectural, design, implementation, and other process aspects. VAX/SVS was designed to meet the requirements for class A1 of the US government's Trusted Computer Systems Evaluation Criteria – the TCSEC or the Orange Book. In addition, we find that the reference monitor concept runs through all of the lessons here from VAX/SVS. Security controls are gathered in a single place, which is always invoked, made tamperproof, and is small enough to validate. A security policy was required, and targeting a multilevel operating system meant choosing Bell and LaPadula as the core policy. Other aspects of assurance covered by the Orange Book 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 who evaluates the assurance, which in the case of the DoD's Orange Book, was an appropriately accredited third party.

In this paper we concentrate on four areas of lessons: 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 controls*, where the access rights to an object are determined by the owner of the object. It was well known that some security problems could not be solved using discretionary access control. For example, the presence of Trojan horses in application software could leak an object to unauthorized users by changing the access rights to that object or writing a copy of that object to another object accessible to the unauthorized users.

An alternative approach to access control is called *mandatory access control*, where the access rights to an object may be constrained by a system administrator. A major goal for high-assurance systems design is to develop an approach by which lattice security models (such as Bell and La Padula [10]) could be enforced verifiably, satisfying the *ref-*

erence monitor concept [8]. By the late 1980s, no commercial operating systems met these requirements rigorously. While several prototypes of high assurance systems that targeted the reference monitor concept were built in the 1970s and 1980s, poor design choices in these prototypes led to large size or poor performance. For example, the KSOS kernel and the 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. However, unlike prior attempts to build VMM security kernels, the VAX/SVS system was designed "from scratch," which significantly facilitated implementation of the reference monitor concept.

Key to developing a simple, clean design for the VAX/SVS access control system is the small number of the types of subjects and objects. First, the VAX/SVS system provided only two types of subjects: users and virtual machines (VMs). Users access the security kernel via a *trusted path* mechanism, and the security kernel performs operations on behalf of particular users given their access rights as discussed below. Second, the VAX/SVS system provides only four types of objects: devices, volumes, virtualized resources, and security kernel files. VAX/SVS exported dedicated volumes and virtual disk volumes to VMs. VAX/SVS had its own files upon which it controlled access, so only privileged processing could use them.

Subjects and objects are both assigned *access classes* consisting of an integrity class and secrecy class. This approach combines the work of Bell and La Padula [10] for secrecy and Biba [11] for integrity to prevent information flows that may leak objects to unauthorized subjects or may allow modification by unauthorized subjects, respectively. The requirements for A1 evaluation were closely tied to the Bell and LaPadula model for mandatory access control: it was forbidden for lower cleared users (or processes) 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 application of the Biba model to problems of commercial data security, 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 and LaPadula model was minimal. In theory, common programs and read-only databases would be created at high integrity and thus protected from modification – but in reality VAX/SVS did not use the Biba model for its own protection and the authors are not aware that any field test user took advantage of the Biba model.

In addition to access classes, subjects may also be given special *privileges*. Such privileges allowed system users (e.g., administrators) to perform security-critical actions, such as managing the assignment of access classes and allowing the modification of security policy.

Using this model, SVS provided single-level virtual machines. If a user needed to do processing at a higher level while reading lower level information, she would connect to a higher level VM and that VM 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 VAX/SVS achieves the aims of the reference monitor concept. Providing 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 access control and sharing was the virtual machine and virtual disk drive. The VAX/SVS development team felt it had plausible approaches to implementing a usable system despite the coarse granularity of objects.

The VAX/SVS security kernel provided tamperproof execution by making a clean separation between the security kernel and the subjects (users and VMs). All the trusted code in the VAX/SVS system runs in the security kernel, so the designers could focus on protecting the entry points. The only trusted entities in the VAX/SVS system are devices and subjects with privileges. Devices are privileged because several have the ability to perform Direct Memory Access (DMA), which permits them to write to any physical memory address. As a result, devices and their drivers are part of the trusted computing base, although their design was not specifically under the control of the VAX/SVS system designers. Subjects with privileges may modify the lattice security policy and other configurations, which obviously can have profound implications. To prevent vulnerabilities, most privileges are only accessible via a trusted path, so the security kernel can authenticate that a real user is behind the action. Two privileges are reserved for VMs, resulting in the need for these VMs to be privileged. The SVS designers were primarily concerned about ensuring that mandatory security could be enforced. As doing these privileged operations outside the kernel did not violate that goal, the designers preferred offering these minimal abilities over making the kernel larger and more complex.

In addition to mandatory access controls and user privileges, VAX/SVS implemented discretionary access controls (access control lists) on objects. The effectiveness of the discretionary access controls proved to be the only major area of disagreement between the development team and the TCSEC evaluators. The design was optimized around "multiuser virtual machines" – if each user required their own virtual machine, the physical memory requirement for a commercially viable VAX/SVS system would have grown beyond what was available and the design would have had to be modified to demand page virtual machines' memories. In the design chosen, 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. All users who needed to access objects at that virtual machine's access class would share the machine.

As a result of the choice of multiuser virtual machines, it was not possible for SVS to determine with high assurance which individual user took a specific action for purposes of enforcing discretionary access control or collecting an audit trail. The SVS team argued that, given the reality of Trojan Horses, the security gain was not 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 of configuring an SVS system for single-user virtual machines, with the expectation that user organizations would configure their SVS systems for (more efficient and adequately secure) multiuser virtual machines. We note that the tradition of documenting an evaluated configuration with specific security attributes, knowing full well it will be largely impractical, continues today.

Layered Design

The VAX/SVS layered design approach proved to be key to a number of areas. Significant use of layering, abstraction and data hiding was called for at B3 and above of the Orange Book. A levels of abstraction approach in security kernel design was recommended as a means to reduce complexity and an aid to precise and understandable specifications. Reduced complexity was a core principle of SVS; the team lived by Keep It Simple, Stupid (KISS). Other classical layered design principles the team followed included 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 can call any of the lower layers. Lower layers are never allowed to call higher layers. The total number of (potential) interactions in the system is conceptually bounded and restricted; a layer may only call a lower layer, and since each layer defined its external API, only through one of the defined entry points. For performance reasons we did not enforce "no op" calls through intervening layers (though we did briefly consider it). Today, discussions of layered design tend to introduce additional complexity by allowing a richer tree of objects. SVS was an almost pure sequence of single layers (see Figure 1, from [1]). The absolute simplicity of the layering of the system is what gave the layered design some of its power as a structure for both call flow and overall system organization. The cohesion of a layer was based on functionality that naturally needed a great deal of shared code or concepts.

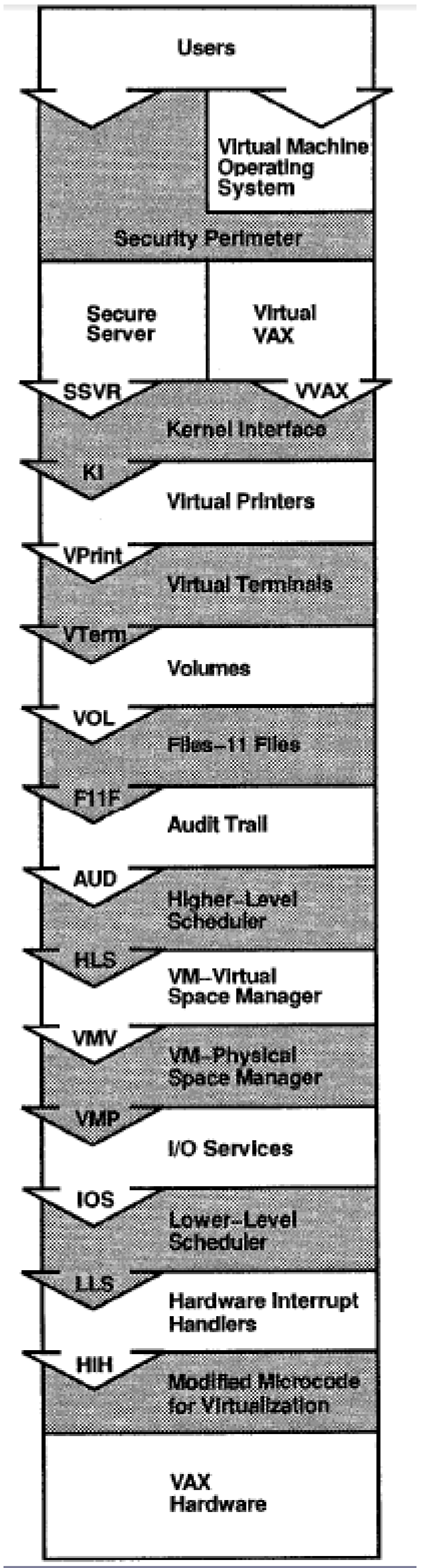

Figure 1 – VAX/SVS Layers

Another benefit of layering is the ability to more easily test a layer in isolation, since the entry points are well defined, and a test environment need only stub out all (lower) layer entry points (at most). Layering damps the overall effect of code changes in the system, and enhances the stability of the interfaces. The grouping and structure that aids understandability can therefore help maintainability (a potential benefit we did not get a chance to see). Reuse, another potential benefit of layering, was not a concern. The layers were designed for the single system.

Testing emphasized layer entry points; what they would do, and what assumptions they made. Part of the layered design rigor was to define the assumptions an API made, and to explicitly check those assumptions “first thing” (assuming they were security relevant). This meant the design of each layer’s surface 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 the 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.

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

Conway’s Law [13] states “… organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations.” This is read to indicate that the interface structure of a software system 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 was built up over time. Potential stakeholders (and reviewers) for design and code changes were the layers above the one changing (or team members within the layer changing if the change did not affect the interface). This seemed unremarkable at the time. If Conway’s Law runs true, the communication structure of a system produced through the agile methodology is likely to be diverse and unstructured, and thus more complex and harder to understand, reason about, and predict.

The counter arguments to a 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. Within the limited experience of a version 0 project, we did not experience any severe constraints on engineering flexibility. Performance and time to market in general are touched on below.

Minimization

Of the three reference monitor requirements, the requirement that the system be small enough to be subject to analysis and tests to assure that it is correct is probably the most challenging for the developers of a secure system. VAX/VMS purported to mediate every access by a subject (process or user) to an object (file, device, or interprocess communication channel) but its size and complexity were great enough so 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.

Choosing to build a VMM, however, was only the beginning of the quest for minimization. From the development project's beginning, there was a conscious intent to minimize the amount of trusted mechanism in the system. Three examples cited below illustrate this point; The first two date back to the summer 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 the initial design study for SVS, and all were veterans of the USAF Multics Guardian project [5]. 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 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 veterans' initial memory management design for SVS included support for demand paging of virtual machines' physical memory spaces. This choice was similar to that of IBM's VM/370. Peter Conklin, one of the original architects of VAX/VMS, participated in the initial design study as a "guest." When he saw the plan to include paging in the SVS kernel, he pointed out that VAX/VMS (which would be running in each virtual machine) implemented paging, that having two independent paging systems was likely to result in poor performance, and that an 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 be feasible to just give each virtual machine a static allocation of physical memory.

The SVS design adopted Conklin's recommendation, which was referred to internally as "memory is cheap." As a result, the VM Physical Memory Management layer was greatly simplified, and the VM Virtual Memory Management layer 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 virtual machine monitors (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. While IBM mainframes implement discrete privileged I/O instructions (that can be made to trap to the VMM), on Digital Equipment computers (PDP-11's and VAXes) 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 developer of a VMM for a Digital computer is faced with the need to intercept every read or write to an I/O register location and to interpret the intended operation. The resulting code is both slow and complex.

The initial design of SVS anticipated virtualizing I/O operations that VAX/VMS would direct toward standard VAX devices. Karger sketched an adaptor that would map virtual machines' I/O operations and minimize software intervention. When Conklin saw that design and considered the problem of I/O from a virtual machine, 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 virtual machines, this would be referred to as "enlightening" the virtual machine's operating system to rely explicitly on the VMM for I/O.

Since 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's version of Unix) was eventually ported to 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 to make VAX/VMS run in a prototype virtual machine monitor 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 a command line rather than a graphical user interface).

After some discussion, the team came up with the idea of parsing administrator commands in an untrusted application running on a virtual machine, and having that application pass the parsed commands (in a simple and standardized format) to the kernel. 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 enabled 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 (no need for confirmation) of commands that would be used frequently by ordinary users 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

Evaluation at Class A1 of the TCSEC was only completed by one or two systems, providing few success stories to emulate. The design of VAX/SVS was extremely simple, even for its day. The team was inculcated with the importance of adhering to the Bell and LaPadula model [10] 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, the verification technology of the 1980s did not follow up on the promises of the 1970s, and the formal verification was confined to analysis of the system's external interfaces against the requirements of the Bell and LaPadula model. 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 in mind. 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 48K SLOC, which were written in PL/1, PASCAL, and MACRO-32 assembly language. As there was over 11K SLOC of assembly code, evaluating assurance was a significant undertaking. The recent formal assurance of the seL4 system [14] shows that the expense per line of code is still high for formal assurance (9,300 SLOC at a cost $10K per LOC). 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 support for devices 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 hardware to commercial processors means that device drivers and their inherent challenges (e.g., controlling DMA) can be moved to 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 the bandwidth of covert channels, so a significant effort was undertaken for the VAX/SVS to eliminate all covert storage channels and mitigate covert timing channels [3]. The simplicity of the VAX/SVS interface and the strict attention paid to adhering to the Bell and LaPadula model resulted in a system that was relatively free from classic storage channels. (Many of the resource allocation mechanisms in VAX/SVS were either static or implemented by human administrators, a set of choices that aided both simplicity and freedom from storage channels.) While 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 UCSB, applied the Shared Resource Matrix method of identifying covert channels, and this work gave the team a rough sense of what channels might be present. The project team collaborated with Robert Morris of NSA to identify an approach to mitigating timing channels – referred to as "fuzzing clocks" or "fuzzy time." Implementing this approach required significant changes late in the development cycle, and the changes degraded the system's (already marginal) performance. Worse, a year after the team published the "fuzzy time" approach, a researcher published an approach to defeating it.

While the VAX/SVS team met the formal verification requirements for Class A1 of the TCSEC, the project team believed that 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). All the security kernel design decisions and code was reviewed at all stages of the project, the code undergoing review before it was checked in. The coding languages for the system precluded buffer overruns, and the style guides adhered to by the team 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 Class A1 of the TCSEC and be sold to customers in sufficient quantity to recover its development costs. By 1989, the system was on track to completing evaluation and sufficiently polished to enter field 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 cancelled because the business case for the system was not sufficient [15]. Even though a great deal of money had been spent bringing the system to the point where customers could use it, sales projections were not encouraging. Some customers were willing to buy some copies of the system, but neither the number of customers nor the number of copies was sufficient to make a profitable business case. United States export restrictions on high assurance products received much of the blame for cancellation at the time, but the reality was that US and other customers who were eligible to buy VAX/SVS were not 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 have been a continuous money-loser.

To understand the reasons why customers didn’t want to buy VAX/SVS, it is only necessary to 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 graphical user interfaces (GUI). 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 was not networked.

Modifying VAX/SVS to support workstations, networks, or graphical user interfaces would have been a significant development task. Workstation support would have been the simplest task though it would have required development and manufacturing of a VAX microprocessor with the SVS-specific virtualization features (which were not included in the standard VAX architecture). And workstation support would not have been viable without adding GUI support. The experience building and evaluating VAX/SVS made it clear that adding networking and GUI support would have been 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 long enough so that by the time the features could be shipped, user expectations would have moved beyond what VAX/SVS could provide. This, of course, was the trap that the decision to build a VMM 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.

Future Challenges

Should We Build Virtualization Kernels Today?

VAX/SVS served two roles in a system: (1) a host security kernel and (2) virtualization management. One question we should consider in hindsight is whether it is practical to require both roles in a security kernel. The VAX/SVS design resulted in a code base of less than 50K lines of code in total. Modern, fully-functional virtualization kernels are much larger still. For example, the Xen hypervisor had about 300K lines of code in 2008. Given the greater size and functionality of modern virtualization kernels and the current cost and complexity of formal assurance, it would seem unlikely that a fully-assured security kernel with fully-functional virtualization could be built today.

Recent hardware advances for security and virtualization may significantly aid the task of separating kernel and virtualization functionality. In adding virtualization support, hardware architects ensure that all sensitive instructions are now privileged, removing the need for I/O emulation. Also, many processors are now self-virtualizable. Finally, and perhaps most importantly, IOMMUs enable DMA devices to be removed from the trusted computing base securely, as mentioned above. With the broadly-available support for trusted computing mechanisms, it is now possible to measure each software layer independently, enabling remote parties to verify the system boot process consisting of multiple layers.

Thus, a question is whether 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 minimization of kernel code. In such kernels, physical resources were partitioned among isolated domains that could communicate through fast IPC primitives. Researchers found such systems to be 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 did not 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 (< 10% in 1997). Further hardware advances have ameliorated the effect of these performance costs, removed vulnerability to DMA attacks, and made it easier to layer software. As a result, it is still an open question when security kernel and virtualization functionality should be combined into a single VMM security kernel.

How Do We Obtain System-Wide Access Control?

VAX/SVS provides a rich access control model that is enforced by a reference validation mechanism to implement the reference monitor concept within the security kernel. Modern systems are not 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). A question is whether it is practical to add a reference validation mechanism at a later time in the system development lifecycle, yet still approach or achieve the reference monitor concept. The emergence of program analysis for security may hold some opportunities for answering this question.

Once a proper reference validation mechanism is in place, a challenge is how to manage the delegation of privilege from the security kernel up to VMs. In the VAX/SVS design, efforts were taken 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. While we have mechanisms to enforce security decisions in privileged VMs, we do not know which software is capable and worthy of being entrusted with those decisions.   Further, this software is far too complex for formal assurance. Finally, while reference validation mechanisms are being added to a variety of software in VMMs, operating systems, middleware, and applications, these individual access control mechanisms are not yet integrated into a system-wide mechanism for managing privilege.

How Do We Assure the Security of Systems?

Fundamental to the VAX/SVS development process was the task of formal assurance. Unfortunately, the task of formal assurance is not much different than it was in the early 1990s. A great deal of manual effort is necessary to convert a system into a format suitable for assurance. In practice, systems cannot be formally-assured unless they are built with that in mind from the outset. The result is a slow, laborious process whose results may be obsolete on delivery.

In developing secure systems, not all software is equal, however, and that may make a difference. Gernot Heiser of the seL4 project [14] conjectured that much of the user-space software would be efficient enough if it was developed in the language used for formal analysis in the seL4 evaluation (Haskell) and then compiled into C. However, the performance-critical software, such as the microkernel, would have to be hand-crafted code. The implication is that formal assurance of the labor-intensive type may only be necessary to a small subset of code that is performance-critical. The remaining code could be developed using compilation tools that would check for security properties. This optimistic view is missing at least two key limitations. First, someone still has to articulate the security properties that must be achieved for the software and for 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 language to improve the security of code or at least enable automated verification without success. Some recent research focuses on making low-level languages amenable to various analyses for security, such as CIL and LLVM. Hopefully, such techniques will be further extended to improve our ability to limit the amount of code that requires manual formal verification.

Conclusions and Summary

We find several lessons from VAX/SVS worth emphasizing and sharing with the broader community today. The Reference Monitor concept from the Anderson report provides useful guiding 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, and raises critical issues as well. Minimization requires whole system thinking, as well as accurate powers of prognostication. Verification of assurance remains a complex (ironically) and multi-faceted challenge. And business realities interact with all of it. The authors believe there are many useful lessons from the VAX/SVS work. We hope and expect they can inform future successful efforts.

Acknowledgements

Each of the authors has fond memories of Paul Karger. Without him, this paper 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 we built.

References

[1] Karger, P.A.; Zurko, M.E.; Bonin, D.W.; Mason, A.H.; Kahn, C.E.; “A VMM Security Kernel for the VAX Architecture”, 1990 IEEE Computer Society Symposium on Research in Security and Privacy.
[2] K F Seiden, J P Melanson, The Auditing Facility for a VMM Security Kernel, 1990 IEEE Computer Society Symposium on Research in Security and Privacy.
[3] J. C. Wray, “An Analysis of Covert Timing Channels”, "Proceedings of the 1991 IEEE Symposium on Security and Privacy", May, 1991.
[4] P. A. Karger and J. C. Wray, “Storage Channels in Disk Arm Optimization”, “Proceedings of the 1991 IEEE Symposium on Security and Privacy", May, 1991.
[5] Adleman, N., J. R. Gilson, R. J. Sestak, and R. J. Ziller, Security Kernel Evaluation for Multics and Secure Multics Design, Development and Certification, Semi-annual progress rept. 1 Jan-30 June 76, Honeywell Information Systems Inc., Mclean Va Federal Systems Operations, August 1976. NTIS AD-A038 261/4.
[6] Neumann, Peter G. and Feiertag, Richard J, “PSOS Revisited”, Proceedings of the 19th Annual Computer Security Applications Conference, 2003.
[7] L. J. Fraim, “SCOMP: A Solution to the Multilevel Security Problem”, IEEE Computer, volume 16, #7, 1983, pp 26—34.
[8] J. P. Anderson, “Computer Security Technology Planning Study”, The Mitre Corporation, Air Force Electronic Systems Division, Hanscom {AFB}, Badford, MA, 1972, ESD-TR-73-51.
[9] Dijkstra, E.W. (1968), "The structure of the 'THE'-multiprogramming system", *Communications of the ACM* **11** (5): 341–346.
[10] D. E. Bell and L. J. LaPadula, "Secure Computer System: {Unified} Exposition and {Multics} Interpretation", Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), ESD-TR-75-306,
March, 1976.

[11] K. J. Biba, "Integrity Considerations for Secure Computer Systems", ESD-TR-76-372, The MITRE Corporation, April 1977.
[12] Larman, Craig (2004). *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley. p. 27. ISBN 978-0-13-111155-4
[13] Conway, Melvin E. (April, 1968), "How do Committees Invent?", *Datamation* **14** (5): 28–31.
[14] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, "seL4: formal verification of an OS kernel", SOSP, 2009, pp 207-220.
[15] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, Clifford E. Kahn, "A Retrospective on the VAX VMM Security Kernel", IEE Transactions on Software Engineers, Volume 17 Iusse 11, November 1991.

Steven B. Lipner
3521 East Alder St, Seattle WA 98122
Phone (425) 705-5082
Fax (425) 706-7329
Email: slipner@microsoft.com

Steve Lipner is Partner Director of Program Management in Trustworthy Computing Security at Microsoft. He leads Microsoft's Security Development Lifecycle (SDL) team and is responsible for the definition of Microsoft's SDL process and for tools and programs to make the SDL available to organizations beyond Microsoft. Lipner is also responsible for Microsoft's corporate strategies and policies for supply chain security and for strategies related to government security evaluation of Microsoft products. He began his work in cybersecurity in 1971, and spent much of his early career leading teams that developed concepts and models for high assurance operating systems. He led the VAX SVS team at Digital Equipment from its inception in 1981 to its cancellation in 1990. Lipner is coauthor with Michael Howard of The Security Development Lifecycle (Microsoft Press, 2006) and is named as inventor on twelve U.S. patents and two pending applications in the field of computer and network security. Lipner holds S.B. and S.M. degrees from the Massachusetts Institute of Technology and attended the Harvard Business School's Program for Management Development.

Trent Jaeger, Associate Professor
Pennsylvania State University, CSE Dept
346A IST Bldg, University Park, PA 16802
Email: tjaeger@cse.psu.edu
Ph: (814) 865-1042, Fax: (814) 865-3176

Trent Jaeger is an Associate Professor in the Computer Science and Engineering Department at The Pennsylvania State University and the Co-Director of the Systems and Internet Infrastructure Security (SIIS) Lab. Trent has a B.S. from the California State Polytechnic Univerity, Pomona in Chemical Engineering in 1985 and M.S. and

Ph.D. degrees from the University of Michigan, Ann Arbor in Computer Science and Engineering in 1993 and 1997, respectively. He joined IBM Research Watson in 1996, and he joined the faculty at Penn State in 2005. Trent's research interests include operating systems security, and the application of programming language techniques to security. Trent is the author of the book "Operating Systems Security," which examines the principles and designs of secure operating systems. He is active in the security research community, publishing regularly, as well as chairing and participating in numerous program committees for security conferences. He is an associate editor with ACM TOIT and has been a guest editor of ACM TISSEC.

Mary Ellen Zurko
230 Nashua Road, Groton, MA 01450
+1 508 254 1389
mzurko@us.ibm.com

Mary Ellen Zurko (aka Mez) is a security architect and strategist for IBM's SmartCloud For Social Business suite of products, which provides collaboration and social networking services. She has led security for projects and products in collaboration platforms, email, authorization infrastructure, web browser and server, rich client, virtual machine monitor, and public key infrastructure. She is active in security standards, having chaired the W3C Web Security Context Working Group, the first standards effort in usable security. She is also 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. Zurko holds a B.S. and M.S. in Computer Science from MIT, both with theses in security. She is the only "Mary Ellen Zurko" on the web; you can find out everything else about her through a web search.