VM-based Security Overkill: A Lament for Applied Systems Security Research

Sergey Bratus
Dartmouth College

Michael E. Locasto
University of Calgary

Ashwin Ramaswamy
Dartmouth College

Sean W. Smith
Dartmouth College

ABSTRACT

Virtualization has seen a rebirth for a wide variety of uses; in our field, systems security researchers routinely use it as a standard tool for providing isolation and introspection. Researchers' use of virtual machines has reached a level of orthodoxy that makes it difficult for the collective wisdom to consider alternative approaches to protecting computation. We suggest that many scenarios exist where virtual machines do *not* provide a suitable tool or appropriate security properties. We analyze the use of virtual machines in the systems security space and we highlight other work that questions the current (ab)uses of virtualization.

The takeaway message of this paper is that "self-protection" mechanisms still represent an interesting and viable path of research. At some point, hypervisors (or whatever the lowest layer of software, firmware, or programmable hardware is) must rely on detection and protection mechanisms embedded within themselves.

Categories and Subject Descriptors

H.1.1 [**Models and Principles**]: Systems and Information Theory—*Value of Information*

General Terms

Security, Measurement

Keywords

virtualization, isolation, VM

1. INTRODUCTION

Virtualization is clearly an enabling technology: providing execution containers can make computing cheaper, more mobile, easier to back up, share, and archive entire OS environments. Many variations on the theme of pure virtualization exist, from open-source CPU emulators like Bochs and QEMU through container-based systems like User-mode Linux, OpenVZ, and Zap [31] to more commercial offerings like VMWare and VirtualPC (and their open-source cousins like VirtualBox). Virtual machine management infrastructure has — notably and recently — increased the practicality of large scale, commodity distributed computing (i.e., the "cloud").

In addition to arguments involving reduced management, administration, and hardware costs, virtualization technology routinely sees service in a security context, primarily driven by the assumption that virtualization provides the best approach (for some value of "best", whether this means (1) "most practical"; (2) more comprehensive than BSD's `jail`, `chroot(2)`, Janus, or systrace [35]; or (3) something else) to providing isolation between execution containers. At an increased cost (typically), a virtual environment can provide inspection capabilities in addition to basic isolation. This kind of isolation and trapping of sensitive operations presents a temptation too sweet for system security researchers to resist (understandably, since pre-existing environments reduce the workload of setting up an execution container).

While we agree that virtualization can be quite useful in many scenarios, we observe that its level of use has approached orthodoxy in terms of the appropriate technology for composing security systems involving isolation, inspection, introspection, or behavioral analysis of execution. In short, as a field, the community seems to have found the perfect implementation form of a practical reference monitor.

1.1 Contribution

This paper attempts to question that orthodoxy. Are VM-based solutions scalable or even economically feasible to enterprise and SCADA network with respect to management and administration overhead they require? Are non-VM approaches still a viable and practical means of achieving isolation, inspection, and other forms of program behavior analysis? Specifically, in situations where sliding another security-enforcing layer such as a hypervisor/VMM might prove too costly for the platform, software might inevitably fall back to examining itself. We suggest that (1) such a fallback is inevitable for certain scenarios and so (2) it has to be done with appropriate engineering principles and care to make it least ad hoc as possible.

1.2 Motivation: Detecting Malicious Computation With Little Performance Impact

Our motivation to examine the possibility of non-VM approaches to software supervision, introspection, mediation, and isolation began with a consideration of how to implement anti-rootkit detection capabilities on a range of low-power and low-resource embedded platforms (e.g., mobile phones, 802.11 access points, SCADA controllers). Such devices typically do not have the resources to run an

NSPW'10, September 21–23, 2010, Concord, MA, USA.

entire VM infrastructure. Furthermore, as a practical matter, using currently available commercial or F/OSS VM technology entails simultaneously running and maintaining two operating systems, with all their attendant services, libraries, and software packages, the locking down and constant patching of which presents a dramatically increased administrative burden compared with our design (a relatively simple, self-contained kernel module that has but one task: monitoring a set of hooks, which we examine in more detail in our technical reports [38, 37]). As explained below, in that system we focus on computation rather than code. In particular, we focus on *detecting violations of an observed, known invariant of the correct Linux kernel operation*.

Rootkit programming Historically, rootkits and exploits tended to be mistakenly associated with malicious, foreign *code* introduced into the system; consequently, anti-rootkit efforts focused on detecting a *foreign body of code* one way or another. This mistaken assumption has been challenged by original hacker research publications [14, 51, 29, 16] that propose flexible exploit programming techniques that involved no foreign code, and has hopefully been defeated for good by [44, 13], which made it clear to the academic community that they were dealing with a *Turing-complete programming and execution model* rather than an ad-hoc collection of obscure hacks.

Hund et al. [22] have further generalized and applied this model to rootkit programming. They stress that the essence of a rootkit was not *malicious code* but rather *malicious computation* effected either exclusively or largely with the unanticipated use of existing, trusted code integral to the exploited system.

We note that the art of co-opting and re-using existing (and therefore inherently more stable) code has been an established pattern of rootkit programming since its early days.[1] In fact, we believe that it would fair to speak of *Rootkit Design Patterns*, such as "context-based hot patching", "multi-layer interception", and others [33, 48, 32], all dedicated to *runtime manipulation of both live code and live data while lowering the probability of a crash*.

1.3 Philosophy on Self-Monitoring and Same-layer Attacks

Essentially, our previous work [38, 37] proposes an alternative to VM-based monitoring that uses a *ring 0* technique for catching a particular class of hostile behaviors by *ring 0* malcode. Any such technique has an intrinsic weakness to defend: an attacker with the capability to arbitrarily modify kernel memory can interfere with it. The VMM/hypervisor architecture has long been accepted as the principled cure for this weakness. We thus face a philosophical question: should we abandon "ring0 vs ring0" techniques entirely, and consider only those that work under the guaranteed protection of the hypervisor? Have such techniques nothing to teach us?

We do not believe so, for two reasons. First, in practice, solutions to software security problems rarely have the luxury of restarting from scratch even when a better design is available and well–understood. In particular, whereas a principled new design would make undesirable behaviors impossible, a practical security solution is constrained to instrumenting existing systems to detect and counteract such behaviors.

A classic example is provided by network stack vulnerabilities: being "immersed" into the protected network, a NIDS stack can be just as susceptible to packet-based attacks as the target systems [21]. Nevertheless, the community does not reject NIDS as a concept on this basis. Likewise, host-based firewalls, (for example, so-called "personal firewalls"), despite their history of vulnerabilities, continue to be adopted.

Second, the study of instrumentation for certain classes of undesirable events, even when undertaken in the presence of theoretically cleaner designs that would obviate the need for such instrumentation, often shows the way toward more efficient and economical designs. Practical OS security engineering offers an an impressive gallery of ideas that started as "hacks" (OpenWall, PaX, LIDS, BSD Jails and Linux VServer) and ended up informing industrial solutions like ExecShield, GrSecurity hardening patches, LSM, SELinux, and AppArmour.

Finally, although one could argue that the components necessary to support VMMs now exist in commodity hardware, using these capabilities entails warping legacy systems around them, *i.e.*, either explicitly rewriting kernels (paravirtualization), or writing device emulation layers: a heavier proposition than instrumentation-based self-analysis. We next consider some additional shortcomings of the VMM-based monitoring philosophy.

2. WHAT EXACTLY IS WRONG WITH VIRTUALIZATION?

As organizations increase their adoption of virtualization environments, and with the current industry focus on information security, it is natural to wonder just how a virtualization framework might pull double duty by improving a security posture as well as easing management burden and infrastructure costs. Of course, merely running a virtualized environment does not automatically entail a guarantee of increased security. As we discuss below, even the basic isolation properties of a VM framework are questionable; it remains entirely unclear whether a VMM can get the job of isolation (a job that OS designers and microkernel researchers have grappled with for years) correct, especially in the absence of hardware primitives for this purpose.[2] We also agree with the sentiment expressed in recent work [6, 11, 24, 40, 17] (some of it our own) that current VM implementations actually use a flawed event trapping framework that fails to capture events of natural and significant interest to security policies.

2.1 "Perfect" Designs

The history of military thought is replete with designs of "overkill" weapons meant to be overwhelmingly powerful or precise, but in reality fall far short of their presumed promise, either in the prototype tests or (much to the wielders' disappointment) on the actual battlefield. Such designs tend to be founded on a crisply formulated, solid, well-established principle or conventional wisdom – right up until reality presents its bill, showing that the accepted theory got to be so neat by ignoring some crucial practical consideration. For example, supertanks (1000 metric tons, 15,000 horsepower) faced problems such as: the weight causing the destruction of roads, the propulsion requirements exceeding the limits of engine power, and easy targeting by air power. As another example, the "Tsar Tank" from Russia, at nearly 30 feet tall and 40 feet wide, the prototype carried 3 cannons and other weaponry, and it was abandoned after unsuccessful 1915 field tests. Finally, the "Tsar Cannon", from 1586 (and on display today) provides an interesting lesson. At 35-inch caliber and nearly 40 metric tons, it is reportedly the largest howitzer ever built. It never shot the 2-ton cannon ball displayed with it; there is no record of it ever being used in battle.

We believe that such a situation is forming in academic research

[1] See, e.g., THC's "(nearly) Complete Linux Loadable Kernel Modules" rootkit HOWTO, 1999.

[2] `http://kerneltrap.org/OpenBSD/Virtualization_Security`

aimed at defeating rootkits – due to a popular misinterpretation of the initial success of VM-based research prototypes.

A series of influential papers [15, 26, 18, 4, 19] (among others) demonstrated the power of running the protected software – including monolithic OS kernels such as Linux and Windows – within a virtual machine, which was capable of intercepting and completely mediating all accesses to all relevant OS objects and data structures.

The underlying host system in its entirety essentially played the role of a *trusted reference monitor* for the guest system. In practice, the host OS is augmented with a reference monitor component loaded with the "protection map" of the guest's internals, derived through either static or dynamic analysis of the latter, or both.

Enforcing a desired security invariant through VM-based complete mediation of a system that can, at any point, be transparently stopped and examined for its violations, is a powerful technique. It appears, however, to have become a "gold standard" of invariant-based policy enforcement research, with anything "less" being considered unworthy of researchers' or readers' time. This view, in our opinion, ignores the practical considerations we relate below.

2.2 Examples of Emergent VM Complexity

The following five points **all possess an underlying theme: that of burgeoning complexity based on forces such as the need or desire to add features or create communications channels.** The first two points deal with the forces driving the addition of features and emergence of complexity in the **"underneath"** layers of a VM-based architecture. The latter two examples discuss the pressure to create complexity **within and between execution containers**, especially as it relates to information flow control. The middle example finishes the set by dealing with the forces driving the creation of complexity at the **interface or boundary** between a VM container and the execution guests within this container.

This point cannot be overstated: central to this paper is the assertion that despite the benefits of VM-based approaches, they have very real challenges derived from the forces driving an increase in their complexity, *but these challenges are routinely ignored* when considering VM-based approaches in the abstract as the principled cure for a security problem.

1. Reliance on virtual machines does not take into account the need for remote management of protected platforms.

Many kinds of systems (most prominently embedded systems but also "headless" compute cluster systems) require remote management. Embedded systems based on commodity software need protection as much or more that desktops (e.g., [2, 9]).

In fact, as Dan Geer convincingly argued [20], the design choice between equipping an embedded system with a remote management interface or releasing it without one is an *evolutionary* choice – whereas remote management interfaces increase the attack surface and risk remote exploitation, the opposite choice often means the loss[3] of an entire released generation of devices to a change in their deployment environments or to a new unanticipated attack. Clearly, leaving the VM environment without a remote management interface is unlikely to be a popular option.

With the importance of remote management in mind, VM-based rootkit protection of a guest OS presents considerable challenges that, in our opinion, significantly reduce its appeal. *If the host OS is to be remotely managed, where should the management network stack (including the management application itself) reside?*

[3] At least, for their control components, which will need to be physically replaced, likely incurring prohibitive labor costs, possibly making the entire product a financial loss.

In practice, remote administration resides on top of a complex environment that includes not only a full TCP/IP stack and HTTP server, but also a CGI scripting platform (which are notoriously hard to secure due to the lack of a well-defined security model).

Interfaces based on command lines and SSH are somewhat better, but still require a full TCP/IP stack and likely also depend on implementations of SSL, DNS, and other standard Internet host functionality – all not only theoretically susceptible to attack, but quite actively exploited in the past.

We can, of course, imagine a highly hardened dedicated control network stack that mitigates most of these issues, but so far economic pressures have unceasingly driven vendors to least-effort commodity software for actual product development and deployment, and this trend is unlikely to change. Moreover, a developer willing to invest all the necessary effort into a *specialized secure VM host* platform to secure a *commodity OS* to support an application might spend the same effort much more profitably developing a specialized non-virtualized platform in the first place!

2. Seeing double: twice the maintenance fun.

Securing a remote management interface to the host system is not the only practical challenge: there is the management itself, such as applying security updates and handling any underlying network changes (which are not infrequent in enterprise environments). In the situation when *both* the protected guest software and the protecting host OS are commodity, security updates will likely be required for both, especially considering the network-facing management functionality of the host system.

With an appropriate automated update infrastructure, the effort needed for doubling the number of maintained systems might not be doubled, even though in the case when the host and guest commodity systems are not the same, *two* infrastructures would be needed to support them.

However, whenever *update testing* is required prior to deployment – an expected procedure in mission-critical environments such as SCADA systems – the labor costs of such testing cannot be substantially reduced. Indeed, even if an organization purchases management services from a third party, that party will incur them anyway for each software item, and will have to pass them to the customer.

3. The temptation of guest-to-host API.

So far, we have considered the host system and the protected guest as separate, non-communicating entities. In such a model, the guest OS and its applications are written to be unaware of running in a virtualized environment, even though this illusion may be neither perfect nor intended to be perfect, i.e., the designer has no goal of suppressing any virtualization-revealing "red pills" [41].

In practice, however, *the symbiotic relationship between the protector and the protected presents an almost insurmountable temptation to developers: that of guest–host communications*.

Indeed, the typical guest's *raison d'etre* is implementing some complex and valuable functionality, which implies that it is likely to undergo some complex and important events or state changes. These events or state changes are best described and detected within the guest's internal logic, but the host system might need to be notified of them – from the guest. The typical purpose involves the export of data formatted by the guest and parsed by the host for administration, management, or auditing. Examples include: (1) sharing of high-value or configuration files across filesystems to avoid the cost of managing several independent copies, (2) migration of process or data from an overly loaded cluster/cloud node, or (3) attempts at guest recovery without losing state during a reboot.

Theoretically, any guest–host interfaces are a bad idea because it breaks the platform's security model by complicating its isolation assumptions. However, the ongoing record of VM escape attacks [47, 27] shows that the temptation of adding such interfaces never wanes despite previous adverse experience, and may even be essential to "cloud" environments.

In the light of the above analysis of practical system management challenges, it should be clear that interactions between the host VMM layer and the protected guest through dedicated interfaces will appear as a tempting way of addressing these challenges. This arrangement, however, will mean passing and parsing of complex inputs, possibly crafted by the compromised guest – with the usual disastrous consequences for the host's security.

4. Information Flow Policy.

The use of virtualization to support isolation is really only half a solution. If system owners take a "deny by default" approach, then they probably have an interest in what safe interactions the virtualization framework *allows*. As Bellovin pointed out in his October 2006 CACM article "Virtual Machines, Virtual Security," [6] containers need to transfer and exchange information. What current virtualization platforms seem to lack is a systematic way to talk about the flow of information across the isolation boundaries between containers. He further pointed out that, even if this capability existed, policy writing is still a hard problem that isn't very amenable to automation.

Note that the general form of this problem (controlling information flow between task units) exists both within the context of traditional operating systems (process, filesystem, and memory space isolation) and in a virtualization environment. Even if a policy framework for controlling information flows between virtual containers (e.g., guest OSs) existed, then all the hard work falls on whoever undertakes the task of policy engineering: specifying the many ways that two or more threads of execution in a fluid set of guests might affect some amorphous set of resources across those guests. The details matter here: the type of events a virtualization system observes has an impact on the performance of the system designed to measure them and make security-related decisions about them.

5. Resource Provider and Reference Monitor.

From a certain point of view, the amalgam of virtualization technology and information security techniques represents a rather strange blend. What does emulation or multiplexing of physical devices have to do with the enforcement of a variety of security properties?

As we mention above, the easiest answer (and the most traditional one) is that virtualization provides an effective means of isolating execution environments; virtualization seems like a natural way to provide isolation between execution containers. Complete isolation, however, is the exception rather than the rule (as Bellovin hints [6]), and customizing the communication between such containers presents a challenge. Thus, even the "obvious" security application of virtualization is fraught with difficulty.

Unfortunately, it appears that little thought has been given to what the best way is to combine the twin roles of resource provider and reference monitor within a single virtualization framework. As a result, virtualization environments can find themselves attempting to measure security-relevant properties of a system in ways that are both creative and convoluted. In essence, the set of events that are interesting from a security viewpoint[4] are not necessarily the set of events that the virtualization framework was built to intercept and observe with a minimal performance impact. Karger and Safford's article [24] details the I/O complexities of most of the popular approaches to providing virtualization.

In this area, we have previously (1) identified the problem of designing an efficient event trapping system of use for both security policy enforcement and virtualization [11] and (2) considered new hardware support for fine-grained information flow labeling and access control that did not involve virtualization [10].

While the suggestion that the design of current virtualization solutions is actually a hindrance to providing security solutions may not sit well with folks interested in touting a particular virtualization solution's security capabilities, we argue that the community has a unique opportunity to make sure that VM platforms are designed to do the things we are asking them to do. Now is also a good time to note that the stunning complexity of VMM I/O subsystems, the performance hacks therein, and the backdoor management interface all suggest that even the basic isolation story rests on somewhat shaky ground — a reasonable basis for suggesting that alternative approaches may deserve some research attention.

We find ourselves at a unique point in time: we can try to identify the right design for doing these two disparate tasks at once, or we can muddle through by abusing a framework meant for resource multiplexing rather than program supervision. In either case, we still must balance the tradeoff between the virtualization frameworkâĂŹs I/O architecture and subsystems and the trustworthiness of the reference monitor. Ironically, as we depend on VM frameworks to implement more security functionality, **these systems become less trustworthy even as they become more trusted**.

3. RESEARCH QUESTION

One major theme that arose from the reviewer's comments on the submission version of this paper was a desire for more direct comparison of the relative performance and efficacy of approaches to measurements of security-related events. Karger and Safford [24] have studied this question, and they conclude that:

> Modern I/O architectures are quite complex, so keeping a virtual machine monitor (VMM), or hypervisor, small is difficult. Many current hypervisors move the large, complex, and sometimes proprietary device drivers out of the VMM into one or more partitions, leading to inherent problems in complexity, security, and performance.

Their article reviews a variety of virtual machine architectures and illustrates both the performance challenges and resulting complexity arising from attempting to deal with I/O management and interception in each virtual machine architecture.

In his 2009 ASIAN keynote [23], Karger goes on to say:

> It is widely believed that the use of a virtual machine monitor (VMM) is at least as secure, if not more secure than separate systems. A recent Information Week survey reports that 55% of responding business technology professionals believe that a system running in a virtual machine is as safe as physical servers and 20% believe it safer than physical servers...in reality, the security of a single system running in a virtual machine can never be as secure as that single system running in its own dedicated physical hardware...because there are more lines of code that must be correct, the VMM case always has more opportunity for exploitable flaws.

[4]And this set depends on what type of "security" you're interested in measuring...from integrity of control flow or data items to information flow to authorization and access control.

3.1 Future Experiments

One reviewer in particular suggested taking measurements similar to those that Autoscopy makes [38, 37] from a VM; this essentially requires re-implementing Autoscopy in some VM framework. We agree that this measurement would help support our assertions about Autoscopy performance and intend to conduct such an experiment in a separate paper on Autoscopy itself.

It would be of interest to conduct experiments that evaluate the relative performance impact of several different approaches (such as those reviewed by Karger and Safford in their article) to virtual machine security — keeping in mind that some of these architectures are used with different security goals in mind (e.g., securing the hypervisor from guest OSs, securing a guest OS from vulnerable applications, securing guests from each other, securing a VM host from another VM host). We did not conduct such experiments during the submission of this paper, although we agree that they would be a valuable basis for further research in this area. Whereas Karger and Safford provide a model-based argument for why performance and efficacy might degrade, their argument can be supplemented with an experimental evaluation of different scenarios.

We plan to conduct experiments dealing with the energy costs of different secure programming primitives and programming models. In essence, whatever the CPU has to do to interpret and enforce a security policy is ultimately a computation. Since a policy is a computation realizing a particular security model [42], the question of cost of security should be posed in terms of this computation (i.e., the nature of security events and state measured by this system).

Most research papers (see Section 6) seem to claim either very low performance overhead ($< 10\%$) or a significant (order of magnitude or more) slowdown, depending on the invasiveness of the instrumentation. Direct comparisons might be difficult to draw, however, given the sometimes differing goals of these systems.

For example, a comparison between a system like Autoscopy [38, 37] (2..5% overhead) and a system like SecVisor [43] is hardly apples-to-apples; in one instance, self-protection of a commodity OS on top of bare metal has almost negligible cost, whereas a hypervisor concerned with protecting the kernel integrity of guest OSs has a performance impact that exceeds 100% (in relation to Xen – and so, even slower than bare metal) in some cases (see Figure 12 in the SecVisor paper [43]).

In any event, our arguments in Section 2 focus on additional costs beyond performance concerns. In other words, we believe that self-monitoring has a substantial performance edge, but for those concerned that it may be vulnerable to attacks or subversion, it has several other advantages over different forms of VM-based monitoring. *Even this way of framing the question is not quite satisfactory to critics, who naturally want both zero performance impact as well as impregnable security and who seem ready to settle for VM-based solutions that are neither zero performance impact nor provide anything close to unassailable security.*

3.2 Challenges for Self-Monitoring

Two main challenges seem to exist in the space of protecting low-level or highly-privileged code. First, the embedding of self-monitoring must in some way attempt to protect itself from malware (rather, malicious computation) executing at the same level of privilege. Thus, self-monitoring systems (like Autoscopy), in order to gain acceptance from the community, must provide an argument as to why malicious computation will find it difficult (i.e., genuinely hard or impossible, not just laborious) to (1) detect, (2) disable, or (3) blind self-monitoring. In essence, *good* self-monitoring instrumentation should be looking at execution events or data artifacts which malicious computation **must** use or modify in order to gain control.

Second, a race exists here: the detection granularity depends to a certain extent on performance considerations; should malware be able to use or modify critical state before detection occurs, then malware has a potential avenue of attack — it has established itself at the same level of privilege. Depending on the architecture of the system, however, malicious computation might have to perform additional actions (which may or not be noticed by defensive instrumentation) to succeed in disabling the monitoring mechanism itself. Self-monitoring should have some way of controlling this race if it is to be effective.

4. REVIEWER COMMENTS

For ease of presentation, we have summarized some of the reviewer comments and our responses to them in this section rather than burying them throughout the paper. One major theme — that of comparison between approaches to security measurement — we address in Section 3. The reviews contained three suggestions, which we term "ease of management", "interface width", and "taxonomy" and discuss below.

Reviewers noted that focusing on detection rather than controlling malicious computation was counterproductive. We concur in this opinion. One reviewer suggested that an interesting question for NSPW discussion was the extent to which VMs could and should be used to support isolation. We do see a place for virtual machines, but our main point in this paper is to suggest that alternatives have at least as valid a claim.

4.1 Ease of Management

One reviewer suggested that, contrary to our assertions about increased management complexity, one major selling point for virtual machine technology has been a *reduction* in management complexity resulting in the ability to scale up to installations of many thousands of virtual computers, along with the ability to manage easy migration and suspension of virtual machine state. The reviewer even offered a suggestion where VMs served a security purpose: evaluation of patches and then live migration to a patched image. This is a good example, although we note that systems like Ksplice, Minix, and our own Katana [12] system — all examples of hot-patching capabilities that *do not* require virtualization.

We agree that *management* has been one of the key features of the shift toward virtualization. What we are objecting to is the application of virtualization to *security* problems that then introduces additional complexity where perhaps a simpler, non-VM solution would not require this complexity.

4.2 Interface Width

The reviews suggested that one reason why the perception VMs are more secure than "self" monitoring mechanisms exists is that the attack surface between user space and OS kernels is much wider than the interface between a guest OS and a VMM. Although we largely agree with this observation, in Section 2, we suggest that pressure exists to widen this interface. Furthermore, we claim that the attack surface is composed of more than just the guest–host API (it includes the management interface and communications channels between VMs).

4.3 Taxonomy

As an extension of the "interface width" observation, this review suggested that a systematic comparison based on an analysis of different attack surfaces might be a valuable contribution. We agree

that this might be an informative way of understanding the benefits of a particular approach and when to use it.

5. WORKSHOP DISCUSSION

The lively workshop discussion explored different directions and attempted to understand what alternatives to virtualization-based supervision might look like.

Sean Peisert asked if the communications boundary between guest kernels and the host VM (which we termed an "API" and asserted as one of the places that pressure exists to increase complexity) necessarily had to have the formal undertones of a programming API. In general, he pointed out, the key problem stems from the observer effect, in which the host taints the guest in some way. Sean wanted to know if alternative data extraction mechanisms exist. We concurred and suggested that "API" simply provided an easy label for this kind of interaction. We also emphasized that the danger exists bidirectionally: in addition to the host perturbing the guest, the host also consumes input data from the guest as the guest replies to queries for information to support the host's supervision/security decisions about the guest. Anything with a sufficiently interesting parser presents a ripe target for attack. Michael noted that the subtext of Sean's question identified a *significant* semantic gap between the information that virtualization (generally, systems driven by some security monitoring policy) monitors for making policy decisions and information that it can easily, efficiently, and transparently observe. We revisited this point several times in the discussion, and we cannot emphasize it enough here: **hardware companies have designed commodity microprocessors to execute code quickly. They largely neglect to include programmable, efficient, and generalizable event extraction and data aggregation primitives for simultaneously monitoring this execution at speed.**

Someone else observed that since commodity virtualization had "arrived", why not just take advantage of these mechanisms (in essence, grin and bear it). We replied that in certain scenarios, virtualization provides natural advantages. We have nothing against it in these situations, and it sometimes provides the most convenient environment for rapidly implementing a prototype or creating a standard, portable computing environment. We asserted, however, that a space of systems (e.g., low power, SCADA) exists where constraints make the use of virtual machines unattractive. We criticize the ease with which people seem to take advantage of a VM as a reference monitor, even though most of them were designed to serve as resource emulators and hardware multiplexers. Taking QEMU, Bochs, or Xen and modifying it slightly often produces a prototype that is "good enough" for demonstrating some trick or technique. The issue, of course, is that in an applied setting, the attack surface explodes because of the "complexity pressures" we identified.

Certainly, in the research community, using a pre-existing environment that is well supported, extensible (or has code available for modifications), and has a user community presents an almost perfect vehicle for applied systems research. In essence, since the main "research" task does not appear to be the construction of the container or execution environment, but rather the protection technique, researchers tend to neglect the formulation of the environment as a distracting systems engineering task (i.e., why re-invent the wheel) in favor of building the security mechanism within the context of the pre-existing execution environment. We suggest that it is critical to the assurance argument for the system that such low-level considerations be revisited during design.

Richard suggested that if you could randomize the address space and instruction set and get the hypervisor to support decoding and relocation, you could have your cake and eat it too. Michael replied that we would have to be careful in this kind of discussion because there are almost endless combinations of protection techniques, VMMs, guests, and hosts to construct; we need to be careful about what threats we are concerned with (e.g., against the hypervisor? against the host? against each guest? guests against each other?). Michael acknowledged that having the hypervisor's code transformed by ISR [5, 25] might protect the hypervisor against code injection attacks, but this protection would not matter if a guest's browser or mail client was compromised in some fashion.

Michael Franz pointed out that researchers have invented hardware support and mechanisms for a guest to *bypass* the hypervisor's management of page table permission bits, thereby "disintermediating the intermediator!" We feel this is a great example that expresses a need so pressing that people had to invent a way around the overly helpful hypervisor.

Sergey pointed out that the general approach here goes back to our theme of coordinating trapping granularity with analysis power: virtualization can trap access at a fine granularity, but it does so in an expensive way, and it requires code (software) to **break through** layers of abstraction, **aggregate** data and events, and **extract** these constructs back up to the level where it makes decisions about them. An alternative design approach is clamored for by people who insist that more logic go into the page tables and MMU. We have previously published a paper that considered an FPGA-based architecture for allowing some processing logic in the policy to occur in hardware, thereby eliminating some portion of the pressure on performance. Again, the overall design of virtualization's trapping scheme is derived from the need to multiplex resources, not interpret security properties.

Andre asked about how we might consider virtualizing Hardware Security Modules (HSM). We noted that there has been work in this area ("Virtualizing the TPM" [7]). The conversation continued with Andre asking about how to build and configure trusted compartments on the fly, which is an interesting problem.

Someone asked if we could compare the self-monitoring approach of Autoscopy with an emulator running on physical hardware. We concur that one experiment we can do (to support Autoscopy) is compare an equivalent implementation in other environments like an emulator (e.g., Bochs, QEMU) or virtual machine / hypervisor (Xen).

Shamal asked about our target environment (e.g., low power systems, SCADA) and whether we had considered how to justify power costs in water-related SCADA systems (not power, as most researchers focus on). Michael stated that the authors were not water SCADA experts and had no idea of the price points there, but that we are starting by analyzing what this might take on a mobile platform. This question grew from our point about investigating the cost of security in terms of the energy taken to interpret a part of a security policy. Sergey clarified that our focus is not SCADA *power* systems, but rather the actual amount of *energy* taken to interpret a security policy in millions of distributed devices. In short, it costs energy to power a computation, and security-related computation is usually an *extra* computation proceeding in parallel or interwoven with the main computation. On a desktop or server, this drain might be no big deal, but as devices replicate, the cost mounts, and in some SCADA scenarios, we might wind up with megawatts spent on security computation. An attendee noted that cost might be an inappropriate way to validate this hypothesis related to our point about virtual machines being extra complexity and computation. Sergey supplied the example of SELinux as one where the kernel definitely interprets an auxiliary computation (and does not involve virtualization).

Jed raised the point that container mechanisms that do not use virtualization exist, such as Google's Native Client [52] and Linux Vserver (as well as things like Solaris Zones). David followed up on this question and asked about whether we thought using alternative approaches to lightweight virtualization may have merit and whether they were useful. Michael's response was that although these mechanisms provide containers, our initial problem context was the problem of detecting and stopping malicious kernel-level computation. The key research issue going forward is how to answer the question: "how can you have a trustworthy system that has to protect hooks at the level of attack?"

In our previous work, we have discussed how such container mechanisms are suitable for isolation; the key research question is on how to efficiently extend them with inspection mechanisms; despite their vastly different architectures, what they provide now is a "labeling" system whereby some extra context information is embedded in the kernel data structures representing processes or other important resources. Jed's question arose from his experience working with a student to imagine how Unix might be different if it were not a time-sharing system, and virtualization may actually provide a step in the right direction when defining multiple relationships between different systems.

Brian Snow liked our assertion that virtual machines are becoming less trustworthy even as they become more trusted. In essence, virtual machines provide a good example of a typical overdependance on a newfound security mechanism: as these mechanisms see their adoption rates soar, they are pressed into service under conditions in which their security assumptions or guarantees might fail catastrophically.

The discussion came back around to the point of virtual machines being an *attractive nuisance*; an observer noted that virtualization per se is an OK mechanism, but that the problem was all the "other stuff" we have thrown into the virtualization bucket along the way lessens its utility. We asserted that the central problem arises from the double-duty of a virtual machine as both a resource multiplexor and a reference monitor. Because a virtual machine is a piece of software, it is ultimately malleable and relatively easy to turn into a reference monitor — a seductive path of implementation, particularly when creating new mechanisms in research efforts.

As the discussion came to a close, Cormac, among others, asked about our choice of *offensive* weapons for the presentation slides, indicating that defensive images such as fortifications or walls might provide a more precise metaphor. We agree, although pictures of really large cannons can be more exciting than pictures of buried or decayed walls.

6. RELATED WORK

While virtualization serves as an enabler for security solutions, it does not necessarily function as a security provider without some careful thought about the design and management of the systems, processes, and people surrounding it.

Security Applications of Virtualization Projects involving aspects of virtual machines and security range from those that show how a VM or VM framework can provide or enhance security functionality intrinsically to those that use VMs as containers to form part of a larger security system. The former type of project looks at what functionality can be added to the VM framework's code to implement things like access control, trusted computing [8], isolation, malware reverse engineering[5], virus scanning, network content filters, information flow analysis, and anomaly detection.

The latter type of project employs to VMs to provide a convenient disposable container to examine the execution of a guest application or OS. Some examples of this use include opening potentially infected emails [46, 45] or web pages [36], testing out patches or other software fixes, and recording application state for replay [15].

Rootkit Detection Traditional rootkit detection approaches like chkrootkit[6] and Rootkit Hunter[7] tend to look for threat-specific information: either known rootkit binaries or known alterations of system binaries, configuration files, or system state (*i.e.,* a network interface set in promiscuous mode). While these tools provide some assurance against basic kernel or user–level rootkits and provided a vital defense during the advent of such rootkits, their general approach is similar to signature-based virus scanning.

Most current approaches to kernel rootkit detection monitor the "known good state" of some static data. Microsoft's RootkitRevealer[8] compares the results of API calls against on-disk registry and file data to detect manipulation of this data by a rootkit in the middle. Other current approaches examine static portions of the kernel text segment (*i.e.,* kernel code) to detect rootkits that attempt to overwrite existing kernel functions. Such approaches typically check cryptographic hashes of kernel text memory ranges against a whitelist of hash values (a mechanism similar to Tripwire[9]). As one example, Petroni *et al.* [34] suggest using a bus-mastering PCI card to periodically monitor the contents of kernel memory via a DMA-like mechanism. Finally, identifying potential kernel *hooks* (control flow transfer points that might enable a rootkit to interpose on kernel execution) for defensive purposes is one interesting avenue of recent research [50, 53] because it enables guarding these locations.

Control flow integrity (CFI) [1] ensures that runtime program execution paths conform to a Control Flow Graph (CFG) learned via static source code analysis. Petroni and Hicks propose State-Based CFI [30]; their approach validates the data structures in the kernel instead of tracking individual execution branches. This approach allows the inspecting process to be external to the system (*e.g.,* in a VMM or on a PCI card) but opens a window during which an ephemeral rootkit might be deployed. SBCFI can also incur close to 40% overhead on a typical machine running Xen (but a major portion of the overhead is attributed to Xen itself).

Kruegel, Robertson, and Vigna [28] propose the use of symbolic execution to analyze LKMs *before* they are loaded into the kernel. This approach, however, requires a static whitelist of valid memory regions and kernel symbols in order to distinguish malicious and benign LKMs. Building such a whitelist can present a challenge for large, constantly evolving systems.

The NICKLE system [39] implements a shadow memory controlled by a VMM for the entire region of kernel memory in the guest machine. On guest boot, all known authenticated kernel instructions are copied into the shadow memory. At runtime, each kernel instruction fetch is verified by comparing the shadow memory maintained by the VMM with the actual physical memory at that location. Differences indicate the presence of a rootkit. The difficulty with such shadow memory schemes is how to handle LKMs: manual certification of an LKM via code signing does not guarantee the absence of malcode in the LKM. It is possible that the large body of work done to handle untrusted modules applies here,

[5] http://bitblaze.cs.berkeley.edu/

[6] http://www.chkrootkit.org/

[7] http://www.rootkit.nl/projects/rootkit_hunter.html

[8] http://technet.microsoft.com/en-us/sysinternals/bb897445.aspx

[9] www.tripwire.org

but it remains an open problem how to integrate these techniques with current production operating systems.

Paladin [3] divides the system into two *protected zones*. Paladin safeguards these zones from illegal accesses by malware. The *Memory Protected Zone* consists of the kernel hook tables and other static regions of the kernel that need to be protected from rootkits, while the *File Protected Zone* consists of system binaries and libraries that need to be prevented against modifications. Given the specifications of these zones, Paladin uses a VMM to monitor write accesses across the system for validity. Any time Paladin detects an invalid access, it identifies and kills the entire process subtree for the process that invoked the offending write. A Paladin driver resides on the guest machine to facilitate this process and interacts with the monitor to aid detection. More fundamentally, monitoring writes to specified locations does not prevent rootkits from hijacking function hooks within data structures because these locations are *meant* to be overwritten, and malcode can use existing kernel code to execute the overwrite, thus avoiding detection based on the "origin" of an instruction.

One closely related piece of work is the HookSafe system [49]. HookSafe maintains a shadow list of hooks in a protected page; the system is based on the assumption that hooks rarely change (i.e., they are typically read many times during system execution but written only a relatively few number of times).

7. CONCLUSION

We suggest that the systems security community should refrain from preemptively dismissing non-VM approaches to isolation and inspection, precisely because virtual machines offer a tantalizing computing primitive. We argue that other ways of assessing trustworthy behavior and measuring security properties exist, and we examined a case study of one way to provide such a monitoring mechanism at the same privilege level as the malicious computation it tries to detect. Such an approach seems risky – and to those steeped in the comforts of a VM mindset, this new kernel-level self-monitoring mechanism seems wrong. Such a viewpoint says: "but the problem is already solved by putting the kernel and everything above it in a VM!" and dismisses the arguments as to why one might *not* want to depend on a VMM as "handwaving."

With such a viewpoint, we risk avoiding the natural evolution of technical approaches. Indeed, why explore other ways of doing things if one way already exists and "solves the problem"? And of course a new way has to be proven all-around superior before it deserves to be communicated. From a certain point of view, there is always something that appears to "solve the problem." Why Perl when shell and grep are already there? Why use Ruby when you have Perl? The Linux kernel was dead on arrival according to certain sages (monolithic kernels being so "yesterday") and so on.

The community runs the danger of failing to realize that virtual machines, despite their practicality in some situations, are not the sole method of implementing a reference monitor. Furthermore, this mechanism has its own risks, shortcomings, and drawbacks. For example, with a VMM, one must effectively provision *two kernels*, one of which may be difficult to manage remotely except by way of the upper-level kernel it protects, or through something like an extra "OS in the chipset/network card/SMM", and all this extra complexity is assumed to come for free.

Acknowledgments

We appreciate the reviewers' comments and the guidance of our shepherd, Ben Laurie. We tried to represent and summarize the reviewers' comments as best we could and feel that such a pattern (i.e., including review comments and authors' response and analysis) would be a useful technique for the community if generally adopted. We also appreciate the responses and feedback we received during the workshop: we apologize in advance if we mis-remembered or misrepresented anyone's comments or point of view. Thanks also to the scribes for our session, Matt Bishop and Mary Ellen Zurko.

NSPW Justification Statement

This work takes a position that questions an existing paradigm: the use of virtualization to support systems security. Are security and virtualization *really* a natural match? Virtual machines have recently revived for a number of applications. Somewhat troubling to us, virtual machines have also been seen in the systems security community as the most practical form of a reference monitor, and have been pressed into service in any number of ways to provide isolation and introspection.

A Virtual Crutch This paper argues that such a paradigm is injurious to the systems security community: we've become addicted to a crutch that is not suitable for all situations, and it is furthermore unclear that the many forms of virtualization are the correct tool for providing isolation and security. VMs are unfit and cumbersome for security tasks for a number of reasons that we examine in the paper. Virtualization is a crutch: it may add stability, but slows down an otherwise healthy system. One whole problem with the VM-based approach is that they typically ignore the incumbent complexity of managing the VM infrastructure.

The main contribution of this paper is the argument dealing with the appropriateness of virtual machines as a security technique. In other words, the "NSPW" contribution is an exploration of whether VM alternatives are a feasible and useful avenue of research that should not suffer a premature death simply because the collective wisdom assumes VM methods are always appropriate.

8. REFERENCES

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2005).

[2] BAEK, K.-H., BRATUS, S., SINCLAIR, S., AND SMITH, S. Attacking and Defending Networked Embedded Devices. In *2nd Workshop on Embedded Systems Security (WESS)* (Salzburg, Austria, October 2007).

[3] BALIGA, A., CHEN, X., AND IFTODE, L. Paladin: Automated Detection and Containment of Rootkit Attacks. In *Technical Report DCS-TR-593, Rutgers University, Department of Computer Science* (2006).

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

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

[6] BELLOVIN, S. M. Virtual Machines, Virtual Security. *Communications of the ACM 49*, 10 (October 2006).

[7] BERGER, S., CÁCERES, R., GOLDMAN, K. A., PEREZ, R., SAILER, R., AND VAN DOORN, L. vTPM: Virtualizing the

Trusted Platform Module. In *Proceedings of the USENIX Security Symposium* (2006), pp. 305–320.

[8] BERGER, S., CACERES, R., GOLDMAN, K. A., PEREZ, R., SAILER, R., AND VAN DOORN, L. vTPM: Virtualizing the Trusted Platform Module. In *USENIX Security Symposium* (2006).

[9] BICKFORD, J., O'HARE, R., BALIGA, A., GANAPATHY, V., AND IFTODE, L. Rootkits on smart phones: attacks, implications and opportunities. In *HotMobile '10: Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications* (New York, NY, USA, 2010), ACM, pp. 49–54.

[10] BRATUS, S., JOHNSON, P., LOCASTO, M. E., RAMASWAMY, A., AND SMITH, S. W. The Cake is a Lie: Privilege Rings as a Policy Resource. In *Proceedings of the* 2^{nd} *ACM Workshop on Virtual Machine Security (VMSec) held in conjunction with ACM CCS 2009* (October 2009).

[11] BRATUS, S., LOCASTO, M. E., RAMASWAMY, A., AND SMITH, S. W. Traps, Events, Emulation, and Enforcement: Managing the Yin and Yang of Virtualization-based Security. In *Proceedings of the* 1^{st} *ACM Workshop on Virtual Machine Security (VMSec) held in conjunction with ACM CCS 2008* (October 2008).

[12] BRATUS, S., OAKLEY, J., RAMASWAMY, A., SMITH, S., AND LOCASTO, M. Katana: Towards Patching as a Runtime Part of the Compiler-Linker-Loader Toolchain. *International Journal of Secure Software Engineering 1*, 3 (2010), 1–17.

[13] BUCHANAN, E., ROEMER, R., SHACHAM, H., AND SAVAGE, S. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of CCS 2008* (Oct. 2008), P. Syverson and S. Jha, Eds., ACM Press, pp. 27–38.

[14] DESIGNER, S. Getting around non-executable stack (and fix). Bugtraq mailing list, August 1997.

[15] DUNLAP, G. W., KING, S., CINAR, S., BASRAI, M. A., AND CHEN, P. M. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)* (February 2002).

[16] DURDEN, T. Bypassing PaX ASLR protection. *Phrack 59*, 5 (July 2002).

[17] GARFINKEL, T., ADAMS, K., WARFIELD, A., AND FRANKLIN, J. Compatibility is not Transparency: VMM Detection Myths and Realities. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–6.

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

[19] GARFINKEL, T., ROSENBLUM, M., AND BONEH, D. Flexible OS Support and Applications for Trusted Computing. In *Proceedings of the* 9^{th} *Workshop on Hot Topics in Operating Systems* (May 2003), pp. 145–150.

[20] GEER, D. Keynote, source boston conference. `http://www.sourceconference.com/2008/sessions/dan-geer-keynote.html`, March 2008.

[21] HANDLEY, M., PAXSON, V., AND KREIBICH, C. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of the USENIX Security Conference* (2001).

[22] HUND, R., HOLZ, T., AND FREILING, F. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th USENIX Security Symposium* (2009).

[23] KARGER, P. A. Securing virtual machine monitors: what is needed? In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security* (New York, NY, USA, 2009), ACM, pp. 1–2.

[24] KARGER, P. A., AND SAFFORD, D. R. I/O for Virtual Machine Monitors: Security and Performance Issues. *IEEE Security and Privacy Magazine 6*, 5 (2008), 16–23.

[25] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)* (October 2003), pp. 272–280.

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

[27] KORTCHINSKY, K. Cloudburst: Hacking 3D (and Breaking Out of VMware. BlackHat USA, July 2009.

[28] KRUEGEL, C., ROBERTSON, W., AND VIGNA, G. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the* 20^{th} *Annual Computer Security Applications Conference (ACSAC)* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 91–100.

[29] NERGAL. Advanced return-into-lib(c) exploits (PaX case study). *Phrack 58*, 4 (December 2001).

[30] NICK L. PETRONI, J., AND HICKS, M. Automated Detection of Persistent Kernel Control-flow Attacks. In *Proceedings of the* 14^{th} *ACM conference on Computer and Communications Security (CCS)* (New York, NY, USA, 2007), ACM, pp. 103–115.

[31] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the* 5^{th} *Symposium on Operating Systems Design and Implementation (OSDI 2002)* (December 2002), pp. 361–376.

[32] PALMERS. 5 Short Stories about execve (Advances in Kernel Hacking II). Phrack 59–0x05. Team TESO.

[33] PALMERS. Sub proc_root Quando Sumus (Advances in Kernel Hacking). Phrack 58–0x06.

[34] PETRONI, N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pp. 179–194.

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

[36] PROVOS, N., MAVROMMATIS, P., RAJAB, M. A., AND MONROSE, F. All Your `iFRAME`s Point to Us. In *USENIX Security Symposium* (2008).

[37] RAMASWAMY, A. Detecting kernel rootkits. Tech. Rep. TR2008-627, Dartmouth College, Computer Science, Hanover, NH, September 2008.

[38] RAMASWAMY, A. Autoscopy: Detecting Pattern-Searching Rootkits via Control Flow Tracing. Tech. Rep. TR2009-644, Dartmouth College, Computer Science, Hanover, NH, May 2009.

[39] RILEY, R., JIANG, X., AND XU, D. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *RAID* (2008).

[40] ROSCOE, T., ELPHINSTONE, K., AND HEISER, G. Hype and Virtue. In *Proceedings of the* 11^{th} *Workshop on Hot Topics in Operating Systems (HOTOS XI)* (May 2007).

[41] RUTKOWSKA, J. Red Pill... or how to detect VMM using (almost) one CPU instruction. `http://invisiblethings.org/papers/redpill.html`, November 2004.

[42] SCHNEIDER, F. B. Enforceable security policies. *ACM Trans. Inf. Syst. Secur. 3*, 1 (2000), 30–50.

[43] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), ACM, pp. 335–350.

[44] SHACHAM, H. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security* (2007), pp. 552–561.

[45] SIDIROGLOU, S., IOANNIDIS, J., KEROMYTIS, A. D., AND STOLFO, S. J. An Email Worm Vaccine Architecture. In *Proceedings of the* 1^{st} *Information Security Practice and Experience Conference (ISPEC)* (April 2005).

[46] SIDIROGLOU, S., AND KEROMYTIS, A. D. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security* (June 2003), pp. 220–225.

[47] SKOUDIS, E., AND LISTON, T. Virtual Machine Security Issues. SANS FIRE Conference, July 2007.

[48] STEALTH. Kernel Rootkit Experiences. Phrack 61–0x0e.

[49] WANG, Z., JIANG, X., CUI, W., AND NING, P. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the ACM Conference on Computer and Communications Security* (2009).

[50] WANG, Z., JIANG, X., CUI, W., AND WANG, X. Countering Persistent Kernel Rootkits Through Systematic Hook Discovery. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)* (2008).

[51] WOJTCZUK, R. Defeating solar designer non-executable stack patch. Bugtraq mailing list, 1998.

[52] YEE, B., SEHR, D., DARDYK, G., CHEN, B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy* (2009).

[53] YIN, H., LIANG, Z., AND SONG, D. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of the* 15^{th} *Annual Network and Distributed System Security Symposium (NDSS)* (February 2008).