Homogeneity as an Advantage: It Takes a Community to Protect an Application[*]

Linda Briesemeister, Steven Dawson, Patrick Lincoln, Hassen Saidi
SRI International

Jim Thornton, Glenn Durfee,[†]Peter Kwan
PARC

Elizabeth Stinson, Adam J. Oliner, John C. Mitchell
Stanford University

Abstract

We examine how to turn the scale of a large homogeneous software deployment from an operational and security disadvantage into an advantageous application community that can detect, diagnose, and recover from its own operational faults and malicious attacks. We propose a system called VERNIER that provides a virtualized execution environment in conjunction with collaborative diagnosis and response functions using a knowledge-sharing infrastructure. We report on the preliminary implementation of the system, its experimental evaluation, and lessons learned during development.

1 Introduction

Consider a deployment of hundreds to thousands of end-user desktop systems, all running the same commodity operating system and office application suite, that is attacked by a self-propagating e-mail worm. The worm is a previously unknown exploit of the e-mail application, so signature-based anti-virus software is unable to detect and stop it. The VERNIER system takes the idea of so-called application communities to provide an end-to-end solution to this scenario by

- Detecting the abnormal behavior induced by the worm on the e-mail application
- Restricting the worm's propagation, thereby limiting the number of infected systems
- Diagnosing the corruption on the infected systems
- Surgically repairing the damage done by the worm
- Inoculating the community to prevent reinfection by the same malware

[*]Approved for Public Release, Distribution Unlimited (DISTAR Case 15533)

[†]Glenn Durfee is now affiliated with Google Inc.

An application community (AC) is a large network of computer hosts in a single administrative domain with similar hardware and software configuration running a common set of applications. Application information such as configuration data or execution traces can be pooled from community members for analysis—for example, in response to a subset of the community becoming afflicted with an unknown ailment. The commonality across the AC enables isolation of potential root causes underlying the ailment, whereas a community of diverse, heterogeneous hosts would not have enough in common to isolate particular variables as potential problem sources.

The goal of the VERNIER system is to turn the size and homogeneity of the application community into an advantage by converting scattered deployments of vulnerable systems into cohesive, survivable application communities [13] that detect, diagnose, and recover from their own failures as well as outside attacks.

We aim to provide a security and dependability framework that is both comprehensive and adaptable. The security community builds excellent tools for very specific problems, such as malware detection, and the dependability community builds excellent tools for problems such as configuration debugging and crash recovery. For complete coverage, we require a system that allows flexible integration of a wide variety of such tools in a way that permits them to cooperate and coordinate. We need an adaptable system to enable us to cope with the continually evolving problem landscape. Novel solutions arise to address previously unseen problems, and IT administrators need a low-overhead way of adopting and integrating these new technologies.

A typical enterprise network will fail in many different ways, due to either accident or malice. Insofar as there is not a single method that can detect all such failure types, there is great value in the ability to integrate specialized detectors. VERNIER's ability to plug in multiple different detectors raises the community's sensitivity to failures. As new software introduces new failure modes, and attackers' methods and goals shift over time, we can update the system to incorporate new detection techniques.

Our contributions are in four main areas:

1. VERNIER provides an end-to-end, extensible framework to integrate a full spectrum of event detectors, diagnostic capabilities, and recovery responses.

2. VERNIER introduces a collaborative, community-based diagnostic framework that attempts to discover root causes of failures, not just their apparent symptoms.

3. VERNIER provides a highly efficient knowledge-sharing subsystem based on JavaSpaces that requires very little bandwidth.

4. VERNIER taught us a number of unobvious lessons during its design and implementation that we share to guide future efforts in this area.

2 Related Work

Locasto et al. [13] introduce and explore the concept of application communities (ACs). The idea of ACs inspired our VERNIER system, which also improves overall security by leveraging a monoculture of similarly configured applications. However, VERNIER differs in using a variety of host-based detectors, which do not rely on the distribution of workload. Also, our mitigation technique is decoupled from detection, which allows other detectors to be easily integrated into a VERNIER system.

Chung and Mok [3] propose the Collaborative Intrusion Prevention (CIP) framework: a set of hosts is outfitted to detect attacks and generate countermeasures, which are then shared among the participants. CIP shares many desirable features with VERNIER. However, VERNIER has been built and integrates a variety of detectors rather than relying on one class of detectors.

Grizzard [8] presents ways to automatically recover from a compromise on a single host. He introduces integrity-based intrusion detection, which fires when a system deviates from a known good state. VERNIER uses virtualization to separate the monitoring from the applications and embraces the concept of integrity-based recovery in our configuration diagnosis capability.

Schnackenberg et al. [16] describe the Cooperative Intrusion Traceback and Response Architecture (CITRA), which integrates both network- and host-based intrusion detectors. CITRA agents form a community and are administrated through a Discovery Coordinator. CITRA is similar to VERNIER in its holistic approach of integrating various detectors with meaningful responses to provide automated, end-to-end security. However, VERNIER coordinates responses without a central authority and the behavior of the whole emerges from local decisions.

White et al. [23] propose Cooperating Security Managers (CSM) to extend intrusion detection systems to a larger environment using peer-to-peer techniques instead of relying on a central authority or hierarchical structures. In comparison to VERNIER, CSM provides much tighter coupling of the cooperating nodes in that it allows one host to detect and handle intrusions on another host.

Janakiraman et al. [12] propose a network intrusion detection and prevention scheme called Indra that is based on sharing information between trusted peers. It uses a publish-subscribe mechanism similar to VERNIER to disseminate information among the participants although the underlying implementations differ. The paper does not provide more in-depth details on what information is distributed and what rules the plug-ins enforce as Indra is labeled work-in-progress.

Wang et al. [22] introduced state tracing and differencing using a computer genomics database for semi-automated troubleshooting of Windows registry configuration issues. VERNIER builds on this work with sickness detector-based automated collaborative diagnosis.

Dourish et al. [5] built a system that gathers and publishes information about the state of a distributed system. Their gatherers are similar to VERNIER's detectors, and they also used JavaSpaces for information distribution. The System Health project focused on gathering system state information and visualizing it, without the collaborative diagnosis and automated repair as in VERNIER.

Finally, Stakhanova et al. [17] present a taxonomy of intrusion response systems, which we use here to classify VERNIER. In terms of *triggered response*, VERNIER clearly is an active response system as it is trying to minimize damage, repair a compromised application, and prevent attacks on other nodes providing signatures of blacklisted files. The other dimension to group systems is by *degree of automation*—here VERNIER could be instrumented to allow only manual response if the policy dictates so. In our prototype implementation, we have instead chosen to allow automatic response, which can be broken down into further categories. VERNIER has an adaptive *ability to adjust* using our escalation strategy when detection events occur repeatedly. It is proactive or preemptive in terms of *time of response* if the community members' policy trusts the shared knowledge and implements suggested security recommendations (such as signatures or firewall rules). Otherwise, VERNIER falls into the category of delayed response, which is also called *intrusion handling* as opposed to *incident preven-*

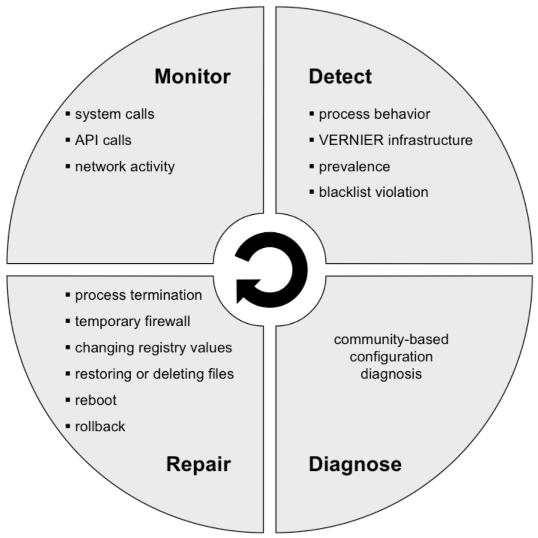

Figure 1: The *monitor-detect-diagnose-repair* loop

tion. The *cooperation capabilities* of VERNIER with respect to response are both, autonomous if our configuration diagnosis cannot provide a solution and the system must resort to reboot or rollback actions, as well as cooperative when the community is actually able to diagnose successfully and suggests a repair for a sick node. In terms of *response selection method*, we classify VERNIER as using a static mapping of alert to predefined response although, as mentioned above, our escalation algorithm does allow different actions based on the occurrence of detections (see *ability to adjust*). The taxonomy emphasizes that for dynamic mapping, certain attack metrics like confidence or severity of the attack must be used to choose a response action.

3 The VERNIER System

Motivated by the idea of Application Communities, we conceived and implemented VERNIER with the following high-level characteristics:

1. Extensible design to easily integrate additional monitors, detectors, and rules for detection and response
2. A monitor-detect-diagnose-repair loop that finds, identifies, localizes, and recovers from abnormal system behavior (see Figure 1)
3. A knowledge-sharing infrastructure through which VERNIER components communicate important events and coordinate activities

Although VERNIER is designed to enable automated protection wherever possible, the architecture is designed to allow for human intervention where necessary.

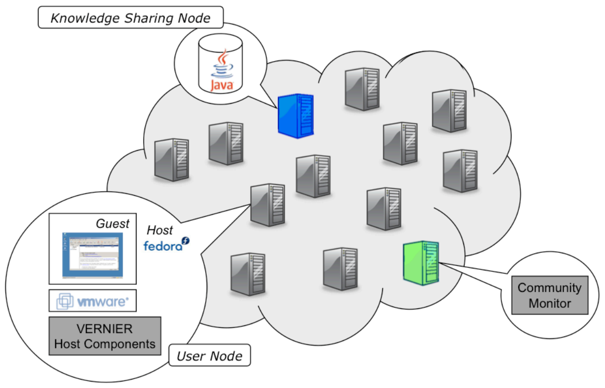

Figure 2: A VERNIER community

A VERNIER community contains a collection of nodes; a node may be an end-user system or a VERNIER component (e.g., a knowledge-sharing or data collection node). Figure 2 depicts a community with a few such infrastructure nodes and many user nodes.

A VERNIER user node contains a collection of host components running in the host operating system, as well as some in-guest components in a guest operating system. Isolating the end-user software in a virtual machine (VM) provides separation from the VERNIER protection mechanisms. The integrity of this separation depends on several factors, including the host hardware and drivers, the virtualization environment, and the in-guest VERNIER components. In-guest components typically include specialized monitors that take measurements not feasibly or efficiently observed from outside the VM and repair agents that must effect changes to the guest. The introduction of in-guest components carries with it the risk of new vulnerabilities; we address this issue in Section 5.

In our prototype implementation, the user desktop systems are all similarly configured with Microsoft Windows XP and the Microsoft Office application suite. Each user system (guest) is contained in a VMware virtual machine environment running on a Linux host, along with a collection of VERNIER components. Figure 3 depicts the logical structure of a VERNIER user node in the prototype system.

A VERNIER node is augmented with VERNIER-specific in-guest components, and a collection of VERNIER host components running in the host operating system. Isolating the end-user software in a VM provides separation from the VERNIER protection mechanisms. The integrity of this separation depends on several factors, including the host hardware and drivers, the virtualization environment, and the in-guest VERNIER components. In-guest components typically include specialized monitors that take measurements not feasibly

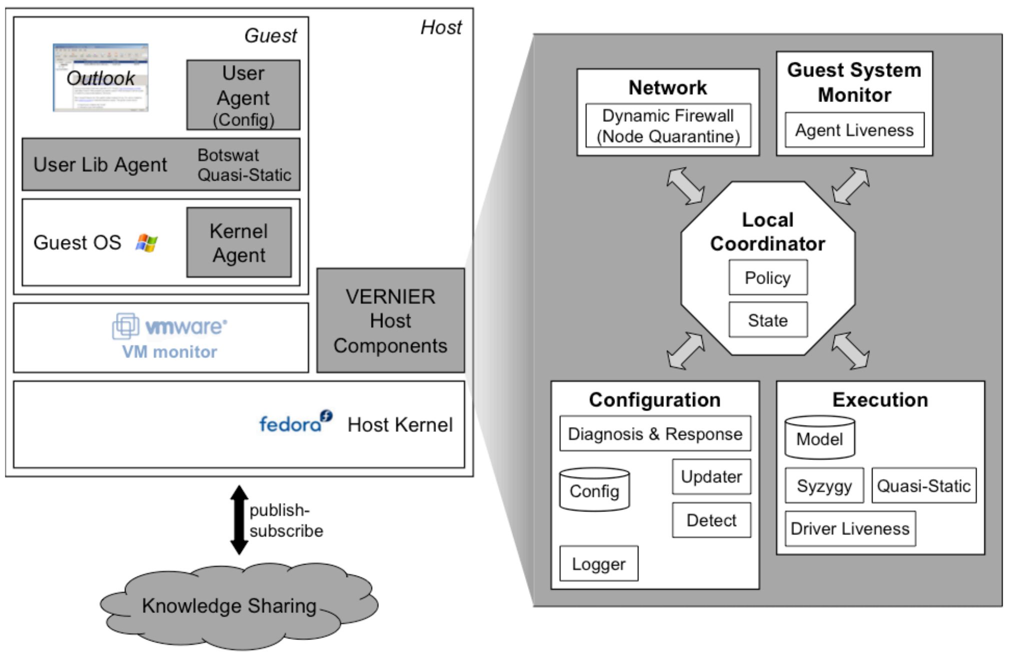

Figure 3: User node with VERNIER components in gray

or efficiently observed from outside the VM and repair agents that must effect changes to the guest. The introduction of in-guest components carries with it the risk of new vulnerabilities; we address this issue in Section 5.

Monitoring involves the observation and reporting of guest system events such as system calls, application programming interface (API) calls, and network activity. Monitoring is typically continuous and is the main driver of other VERNIER activities like detection, diagnosis, and repair. The central tension in selecting monitoring mechanisms is maintaining strong separation of VERNIER and the protected system while monitoring in-guest activity at sufficient granularity to perform useful detection and response. It is easy to inspect the raw memory of the guest OS from outside the VM, for example, but difficult to trace the execution sequence of library API calls in a particular process.

Two detectors focus on the execution behavior of processes in terms of calls into user-level libraries using the STraceNT [6] and Detours [11] call-tracing packages for Windows. These detectors are tightly bound to the monitoring mechanism and execute inside the guest in the process context of the monitored application. This approach enables detection based on the fine-grained behavior of processes with low overhead. We built a custom trace driver for Windows that interposed non-GDI (Graphics Device Interface) system calls at the system service dispatch table [10], selectively copied argument information, and communicated trace records out of the VM to a companion reader running on the host. We used shared memory for the communication channel, with the reader accessing kernel memory pages directly. In the future, we expect to replace this with a VMM (virtual machine monitor)-supported communication channel such as VMCI [20]. Verbowski et al. [19] present a continuous tracer of persistent state interactions in Windows systems and a lossless, highly compressed log format; similarly, VERNIER stores a compressed log database on the host.

We have implemented or adapted a number of host-based and distributed detectors that identify malicious or problematic behavior of the guest operating system or with respect to an application of interest running in the guest.

BotSwat [18] is a behavior-based malware detector that targets the command-execution behavior of malicious bots on their infected hosts. It tracks data received over the network as a program processes this tainted data. BotSwat identifies execution of a bot command when a process uses tainted data in a system call argument. BotSwat detects bots from families that together constitute 98.2% of known variants [14]. Moreover, BotSwat has very low false positives since it is able to identify locally initiated actions, i.e., those behaviors performed by a program at the behest of its local user.

Quasi-Static analysis is an anomaly detection technique that combines static and dynamic analysis to build a model of an application in the form of sequences of user-level Windows API calls. The model is built by static analysis of arbitrary Windows executable files and identifying unknown jump and call instruction targets in these files. The detector looks for deviations from this model.

Syzygy is an *epidemic detector* that looks for time-correlated anomalies in a homogeneous software community—precisely the behavior that would accompany an exploit as it spreads among a set of nodes. An epidemic is the execution of malicious code on a subset of the community. Instead of attempting to decide whether process behavior on a single node is healthy or infected, Syzygy considers the aggregate behavior of a community. Deviations from the average behavior suggest that one or more members of the community are infected.

The **Prevalence Detector** identifies the spread of new and unknown *features* such as registry keys and file names through the community of nodes, which might be part of a propagating attack. The prototype VERNIER system identifies new and unknown files via system call monitoring (in conjunction with the configuration database described below), and records file prevalence in the tuple space of the knowledge-sharing framework. Each node that observes the appearance of a new, unknown file can independently determine whether the prevalence of that file has crossed a threshold to raise an alarm.

The **Configuration Diagnosis** phase aims to identify the root cause of anomalous behavior. We seek to identify a set of features in the persistent configuration state of the machine that can be manipulated to enact a fine-grained, minimally disruptive repair. A machine's persistent configuration state consists of its files and registry keys. Any successful attack on a machine is either persistent, leaving a footprint in non-volatile storage, or is transient, altering only volatile memory. Transient attacks can be reversed by restarting processes or rebooting the system.

To perform configuration diagnosis, we use focused state differencing between "sick" and "healthy" nodes—essentially, a comparison between the persistent configuration state of presumed compromised and presumed uncompromised machines. Our approach is similar to that in PeerPressure [21]; however, we are looking for deliberate attacks rather than diagnosing non-malicious configuration errors. We assume that most nodes remain healthy,[1] so sick nodes will be in the minority. VERNIER relies on the alarms from detectors to indicate when a node becomes sick, providing both the initial trigger for diagnosis and the classification signal to distinguish sick nodes. The diagnosis algorithm correlates relevant features of configuration state with sickness.

[1]This has been articulated as "the golden state is in the mass" [21].

A monitor keeps a configuration database that records all changes to the monitored features of the persistent configuration state, as well as records of which features are used (open, read, written) by which processes. The configuration database is initialized from the configuration "golden state" of an approved install image. Then, the contributions of other nodes through the community knowledge-sharing facility provide the correlated state differencing inputs.

The heart of the VERNIER host components is called the **Local Coordinator** (LC). It implements the node's strategy to react to detections (events of interest that happened on this node) and notifications (events of interest that happened elsewhere in the community and the node learned about through knowledge sharing). The recovery strategy begins by deploying a temporary quarantine to protect the node from a potentially ongoing attack, blocking all network traffic to and from the guest OS until the event handling has finished. This may take from a few seconds to a few minutes. Next, if we are dealing with a detection that refers to a set of processes, the LC terminates all processes associated with the monitored application. The intent is to minimize damage while maximizing stability for diagnostic purposes. Once the application has been shut down, if the attack is transient or was caught quickly enough that it was prevented from establishing a footprint, this may constitute a completely successful recovery. Next, the configuration diagnosis tries to learn about the attack. Our diagnostic capabilities currently only cover problems that leave a persistent footprint, so the result of the diagnosis may be inconclusive. Otherwise, the LC uses the diagnosis to repair the problem by changing registry values, restoring files, or deleting files by comparison with the golden state. Finally, the LC restarts the application for the user and removes the quarantine of the guest system.

VERNIER also accounts for potentially failed attempts at resolving a problem. We use timing information to distinguish whether a new event pertaining to the same application indicates that a prior repair may not have been successful. If such a new alarm is raised within a short time window, we escalate the response strategy from diagnosis and repair to rebooting and finally to rolling back the guest system. A new alarm could also be a delayed detection, however, from another detector with a different reporting delay; a detector with shorter latency may have already noticed the problem and led to a resolution. To account for such a case, another rule in our policy governs how much time between detections has to pass before they are considered for escalation.

Our implementation of the **Knowledge-Sharing** framework employs the tuple-space concept [7], which uses a publish-subscribe mechanism instead of direct message passing between nodes. This allows nodes to join and leave the community continuously without explicitly managing the group. For our VERNIER prototype, we use the Blitz[2] open-source implementation of JavaSpaces.

4 Experimental Results

We perform our experiments on a dedicated testbed for VERNIER. It contains 10 physically identical servers, which are equipped with 4 CPU cores, 8 GB RAM, and 640 GB disk storage, and are connected via a gigabit Ethernet. The server configuration includes Fedora Core 6 Linux as the host OS, VMware Server 1.0.x as the virtualization environment, and the Java 2 Runtime Environment 1.5.0, along with numerous supporting software packages and libraries, especially the Blitz [1] implementation of JavaSpaces. The host software configuration is centrally managed via the Bcfg2 system [4].

To support concurrent development and experimentation, a physical testbed is partitioned into multiple *logical testbeds*, using standard Unix user account and file system functionality. Each logical testbed constitutes a complete instance of a VERNIER application community. Custom software (called tb) enables users to build and manage their own logical testbeds. Functionality provided by tb includes creation and configuration of VERNIER nodes (each of which represents a desktop system) per physical host, configuration and control of guest VMs, and control of communities.

To help explain how the integrated system functions, we use the following demonstration scenario (introduced at the very beginning in Section 1), which is representative of a class of malicious behavior VERNIER is designed to defend against.

A community of 20 user desktop systems, all configured with the same COTS operating system and office application suite, is attacked by a self-propagating e-mail worm. The worm is a "zero-day" exploit of a previously unknown vulnerability in the e-mail application, so standard defenses, such as anti-virus software, are unable to detect and stop it. In this scenario, VERNIER is able to restore the community to normal operation in less than three minutes from the onset of the attack, with no human intervention.[3] Successful community defense in this scenario does not require the full complement of detectors and response mechanisms available in the current prototype implementation. For this demonstration the following VERNIER components are used.

- *Detectors*: Syzygy — detects temporally correlated anomalous application behavior
- *Configuration diagnosis*: tracks changes to files and registry settings connected to the execution of monitored applications
- *Responses*: Dynamic firewall — used to temporarily block traffic associated with an application detected as misbehaving; Process termination — kills processes associated with a misbehaving application; Fine-grained repair — restoring registry and filesystem state based on the results of configuration diagnosis; External blacklisting — augmenting the signature database of installed anti-virus software with signatures generated by configuration diagnosis.

In addition, all the common VERNIER infrastructure components are used; these include the in-guest agent, the local coordinator, and the knowledge-sharing infrastructure. Figure 4 illustrates step by step how the VERNIER components operate to defend the community in this attack scenario:

1. *Initial attack*. The attack begins with the delivery of a malicious e-mail message to one user in the community. The malicious e-mail installs malware into Outlook and restarts it, without any user action. The Syzygy monitor on that node detects anomalous behavior in the application and reports it to the central Syzygy server. No action is taken at this point, since the anomaly has been reported on only one node and might be an innocent transient event.

2. *Detect epidemic*. Meanwhile, the malware uses e-mail addresses found in the address book on the infected node to begin propagating to other nodes in the community. As the malicious e-mail spreads, other nodes begin observing anomalous behavior and reporting it to the Syzygy server. Since it is highly unlikely that this correlated anomalous behavior would occur by chance on many systems, the Syzygy server declares an epidemic, triggering defensive and diagnostic activity across the community.

3. *Contain infection*. Community defense policy directs VERNIER to invoke a dynamic firewall response that temporarily blocks e-mail traffic (SMTP and POP) to and from all nodes. Although this effectively disables e-mail functionality for users, it also has the beneficial effect of limiting the spread of the malware.

4. *Diagnose*. Upon completion of the initial defensive responses, VERNIER begins collaborative configuration diagnosis, comparing changes to the persistent state of infected nodes to the state of uninfected

[2]http://www.dancres.org/blitz/

[3]A video presentation of VERNIER's realtime operation in this demonstration scenario can be found at http://www.csl.sri.com/projects/vernier/demo.html.

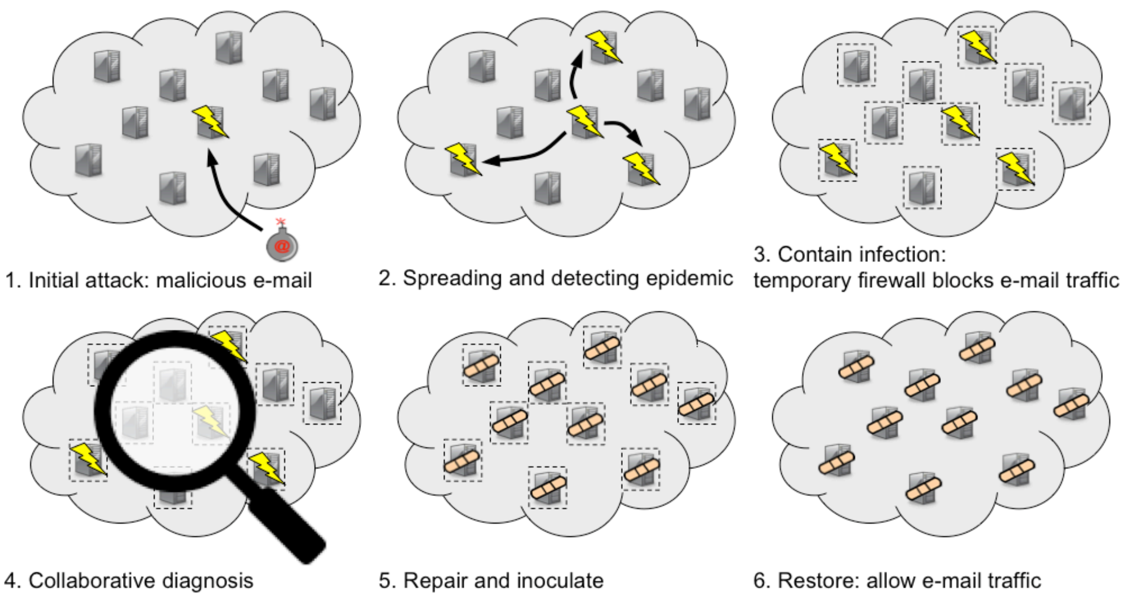
1. Initial attack: malicious e-mail
2. Spreading and detecting epidemic
3. Contain infection: temporary firewall blocks e-mail traffic
4. Collaborative diagnosis
5. Repair and inoculate
6. Restore: allow e-mail traffic

Figure 4: Step-by-step illustration of VERNIER responding to e-mail attack scenario

systems to derive a feature set describing the persistent changes possibly attributable to the malware.

5. *Repair and inoculate*. Using the set of persistent changes identified by diagnosis, VERNIER removes the malware from the infected nodes and distributes a signature of the malware throughout the community; this signature is added to the database of the anti-virus product installed on each node, enabling it to identify and thwart future attempts to install the same malware.

6. *Restore*. Finally, VERNIER removes the temporary restrictions on network traffic, restoring the community to normal operation.

Although VERNIER has not yet undergone rigorous evaluation, early testing has provided valuable insight into VERNIER's performance and the strengths and weaknesses of the approach, which will help guide further research. To gauge the feasibility of VERNIER from a performance standpoint, we have exercised the system through various attack scenarios in small communities (10 to 30 nodes), all based on the e-mail application, and measured the following performance attributes: application slowdown, network traffic overhead, and recovery time.

Application Slowdown A coarse measure of application slowdown was obtained by comparing the elapsed time required to complete an automated e-mail workload with and without VERNIER running. The automated workload simulates very intensive use of the e-mail application (more than any human user could possibly achieve) to reduce skew due to human response time and idle time. Measured in this way, application slowdown due to VERNIER is approximately 10 percent. This is an upper bound on the user-experienced slowdown, although the performance impact of VERNIER on specific parts of application execution may be greater. This strongly suggests that VERNIER's impact should be tolerable.

Network Traffic Overhead VERNIER uses the network both for knowledge sharing and for reporting application execution anomalies from individual nodes to the central Syzygy server. In its quiescent state while not dealing with a detected problem, VERNIER's use of the network is limited to small, periodic node status reports, and averages less than 1 KB/s per node. VERNIER network traffic overhead is highest during collaborative diagnosis, when a significant volume of node configuration state is shared among the community. During these peak periods, which have been observed to last from 1 to 5 minutes, average network traffic due to VERNIER ranges from 10 KB/s to 50 KB/s per node.

Recovery Time An important measure of the effectiveness of a system like VERNIER is the amount of time it takes to restore the community to normal operation once a problem has been detected. For the attack scenarios examined thus far, recovery time has ranged from 2 to 9 minutes, with the majority of the time spent in collaborative diagnosis. Although we cannot currently claim that these recovery times would be typical in a deployed system, our early results demonstrate the benefit of an automated diagnosis and recovery system as compared with manual recovery, which might take hours or even days.

5 Hardening VERNIER

VERNIER is a complex system that adds many new components to the (already complex) systems it is designed to protect. This fact inevitably raises concerns about the potential for new vulnerabilities introduced by VERNIER and whether VERNIER's benefits outweigh the possible risks. Some of the risks arise from known weaknesses in the current prototype that can be addressed by well-established techniques; for example, VERNIER message traffic needs to be encrypted to ensure confidentiality and integrity. Other risks arise from the integration of in-guest components and from VERNIER's automated response capabilities that can effect changes to the guest system configuration; addressing these risks requires more study.

In-guest Components Ideally, VERNIER would not rely on in-guest components at all, and to the extent possible, we plan to minimize their use. However, the ability to monitor guest activity and effect necessary configuration changes and repairs may be either infeasible or inefficient from the host. In such cases, techniques such as *multi-shadowing* [2] may offer a solution. Multi-shadowing is a virtualization-based technique that enables strong protection of memory objects inside a virtual machine, and could be used to ensure the integrity of VERNIER in-guest components. As VERNIER already employs virtualization to isolate the majority of VERNIER host components from the guest systems in the application community, the application of a technique like multi-shadowing would be a natural extension.

Automated Response Another concern is the potential for misuse of VERNIER's automated response mechanisms. For example, an attacker that is unable to obtain a privileged foothold from which to do significant damage might instead employ otherwise benign activity to simulate conditions that lead VERNIER into a disruptive response. Such a response could conceivably cause termination of an application process, the removal of critical application files, or reboot or rollback of a user's system, possibly resulting in lost work or data. Part of the answer to this concern lies in the planned incorporation of human-in-the-loop capabilities that will alert users or system administrators to detected problems and proposed remedies, possibly suspending guest system execution temporarily to allow the user or administrator the opportunity to intervene or to select a course of action, along with a checkpoint capability to allow for recovery of data in the event of a critical response failure.

6 Lessons Learned

Our early experiments support the thesis that leveraging shared knowledge can be an effective strategy for coping with failures and malicious activity in homogeneous software deployments. More work is needed to determine whether and how these results can be generalized and applied at larger scales and in real-world settings. Lessons learned from our experience to date reveal several challenges that need to be addressed.

Application monitors need to look for evidence of correct behavior as well as indicators of bad behavior. VERNIER's current suite of detectors is geared toward monitoring applications for indications that incorrect code is being executed. Of course, there are failures and attacks such as denial of service that impair application execution without altering the executed code in any way. Absence of good behavior can be just as good an indicator of trouble as presence of bad behavior.

Detectors should provide diagnosis with richer problem reports. The current BotSwat, Quasi-Static, and Syzygy detectors are effectively binary detectors: they indicate when the processes they monitor exhibit signs of incorrect behavior. The starting point for diagnosis, then, is simply an identifier of a misbehaving process. However, detectors may often be well positioned to make an initial characterization of the problem, which should help make diagnosis more effective, efficient, and precise.

Diagnosis should be extended to consider more dynamic state. Configuration diagnosis currently examines persistent features (registry entries and files). For attacks that leave no persistent footprint, diagnosis will come up empty. To some extent VERNIER mitigates this problem through its response escalation strategy: if a problem persists, VERNIER invokes increasingly more aggressive responses to try to clear the problem. However, this can leave a significant window of opportunity for an attack to spread and interfere with large segments of the community. To improve VERNIER's effectiveness in the face of attacks with no persistent effects, diagnosis needs to be extended to examine parts of the system's dynamic state.

Correlating evidence from multiple vantage points may help. With the exception of Syzygy, which correlates event streams from multiple nodes, VERNIER's current detectors examine behavior local to a single node. It may be possible to strengthen both detection and diagnosis by correlating multiple types of evidence collected from different vantage points. For example, evidence of bot-like behavior observed by BotSwat on individual nodes could be correlated with evidence collected by a network-based bot detector such as BotHunter [9]. Correlating evidence from multiple detectors at different vantage points could significantly reduce the risk of false positives from individual detectors. Also, evidence from one detector could be used to drive a targeted search for evidence by a different detector to provide better input for diagnosis.

Straightforward translation of low-level execution models between operating systems may not work. The event model that drives the Syzygy correlated anomaly detector is based on Unix system calls. For the VERNIER prototype we adapted this model to the Windows environment. The model is built via training on live application execution, and on the Windows platform we observed a lack of full convergence in the training model as new system-call sequences would appear in the generated model, no matter how long training continued. The result was a high false-positive rate for individual Syzygy detectors that had to be addressed by tuning various detection parameters, which ultimately reduced the detector's effectiveness. An analysis of this problem is beyond the scope of this paper, but it is clear that additional work is needed to develop a call-sequence model for Windows that will be as effective as similar models for Unix.

Evaluation of a system like VERNIER is challenging. VERNIER's scope is broad; the system is a complex integration of complementary technologies, and it aims to protect large networks of desktop systems running interactive applications. These characteristics make it challenging to evaluate the end-to-end effectiveness of the system and even to define useful measures of effectiveness. Traditional measures, such as false-positive/false-negative rates used to characterize detector performance, and failure rates used to characterize reliability, may be useful for individual VERNIER components but do not readily apply to the system as a whole. Indeed, the question of evaluating the combined security and dependability of systems is still an active research topic [15]. In our early experiments, we have used various scenarios to enable performance measurements and to identify areas of strength and weakness, but characterizing the effectiveness of VERNIER and providing metrics, which allow comparison to other systems, remain an issue.

Developing an integrated, end-to-end system for robust, survivable application communities (AC) is an ambitious goal. VERNIER represents significant progress toward that goal. Although the VERNIER prototype is far from being a deployable system, our early experience encourages continued pursuit of the AC concept, despite the remaining challenges. Moreover, our work to date has given us a solid platform for continued experimentation with new and alternative detectors, diagnosis algorithms, and recovery strategies, easing the path toward full realization of the AC vision.

Acknowledgments and Disclaimer

We thank VMware, Inc. for providing VMware software to support this research, and Narayan Desai of Argonne National Laboratory for assistance in deploying the Bcfg2 configuration management system on the VERNIER testbed.

We are especially grateful for Drew Dean's inspiration and guidance on this project.

This material is based upon work sponsored by the Defense Advanced Research Projects Agency (DARPA) under Air Force Research Laboratory (AFRL) contract number FA8750-06-C-0182. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA and AFRL.

References

[1] http://www.dancres.org/blitz/.

[2] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2008), pp. 2–13.

[3] CHUNG, S. P., AND MOK, A. K. Collaborative intrusion prevention. In *WETICE-16* (June 2007), pp. 395–400.

[4] DESAI, N., BRADSHAW, R., HAGEDORN, J., AND LUENINGHOENER, C. Directing change using Bcfg2. In *LISA-20* (December 2006), pp. 215–220.

[5] DOURISH, P., SWINEHART, D. C., AND THEIMER, M. The doctor is in: Helping end users understand the health of distributed systems. In *DSOM-11* (2000), pp. 157–168.

[6] GARG, P. *StraceNT – System Call Tracer for Windows NT.*

[7] GELERNTER, D. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 1 (1985), 80–112.

[8] GRIZZARD, J. B. *Towards Self-Healing Systems: Reestablishing Trust in Compromised Systems.* PhD thesis, Georgia Institute of Technology, March 2006.

[9] GU, G., PORRAS, P., YEGNESWARAN, V., FONG, M., AND LEE, W. BotHunter: Detecting malware infection through IDS-driven dialog correlation. In *USENIX Security-16* (2007), pp. 167–182.

[10] HOGLUND, G., AND BUTLER, J. *Rootkits: Subverting the Windows Kernel.* Addison-Wesley, 2005.

[11] HUNT, G., AND BRUBACHER, D. Detours: Binary interception of win32 functions. In *3rd USENIX Windows NT Symposium* (1999), pp. 135–143.

[12] JANAKIRAMAN, R., WALDVOGEL, M., AND ZHANG, Q. Indra: A peer-to-peer approach to network intrusion detection and prevention. In *12th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE)* (June 2003), pp. 226–231.

[13] LOCASTO, M. E., SIDIROGLOU, S., AND KEROMYTIS, A. D. Software self-healing using collaborative application communities. In *NDSS* (February 2006).

[14] OVERTON, M. Bots and botnets: Risks, issues and prevention. In *Virus Bulletin Conference* (2005).

[15] SALLHAMMAR, K. *Stochastic Models for Combined Security and Dependability Evaluation*. PhD thesis, Norwegian University of Science and Technology, Trondheim, June 2007.

[16] SCHNACKENGERG, D., HOLLIDAY, H., SMITH, R., DJAHANDARI, K., AND STERNE, D. Cooperative intrusion traceback and response architecture (CITRA). In *DISCEX-II* (2001), vol. 1, pp. 56–68.

[17] STAKHANOVA, N., BASU, S., AND WONG, J. A taxonomy of intrusion response systems. *Int. J. Inf. Comput. Secur. 1*, 1/2 (2007), 169–184.

[18] STINSON, E., AND MITCHELL, J. C. Characterizing bots' remote control behavior. In *DIMVA-4* (July 2007), pp. 89–108.

[19] VERBOWSKI, C., KICIMAN, E., KUMAR, A., DANIELS, B., LU, S., LEE, J., WANG, Y.-M., AND ROUSSEV, R. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *OSDI-7* (2006), pp. 117–130.

[20] VMWARE, INC. *Getting Started with VMCI*.

[21] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI-6* (2004), pp. 245–258.

[22] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. Strider: A blackbox, state-based approach to change and configuration management and support. In *LISA-17* (October 2003), pp. 159–172.

[23] WHITE, G., FISCH, E., AND POOCH, U. Cooperating security managers: A peer-based intrusion detection system. *IEEE Network 10*, 1 (January/February 1996), 20–23.