Network-Based Root of Trust for Installation

A network-based system installation method that binds a file system to its installer and disk image, thereby thwarting many known attacks against the installation process.

Joshua Schiffman, Thomas Moyer, Trent Jaeger and Patrick McDaniel
Pennsylvania State University

Data centers and large enterprises often use network installation and monitoring to manage the integrity of their deployed systems. This lets administrators focus on hardening fewer systems and cloning them over the network to a multitude of machines. Once the systems are installed, administrators can use remote monitoring tools and services to automate the detection of system behavioral anomalies, intrusions, and illicit file-system modifications. These mechanisms aim to provide *trusted distribution*, whereby administrators can verify that a system was installed as intended and that nothing was secretly modified after installation.[1] Without trusted distribution, it's difficult to ensure the ongoing correctness of a system at runtime.

Unfortunately, network-based installation introduces new challenges to the already difficult process of verifying system installation. The most common way to initialize a network installation is to load a bootstrap program over the network, thus eliminating the need for installation media such as optical disks. However, the additional network access opens the possibility for malicious parties to compromise installer code or corrupt the disk image in flight. Even after system installation, malicious modifications can compromise security-critical files in ways that are difficult to detect in cases such as configuration scripts that lack a well-known correct state. In the presence of such subversive code and hard-to-verify data, attackers can trick monitoring tools into failing to report a problem. Ultimately, administrators need a method to prove that a system has been securely installed and hasn't been modified since that installation.

To achieve this goal, we propose a Network-Based Root of Trust for Installation (netROTI), an installation method that links a file system to its installer and the disk image used prior to configuration. If administrators trust their installer and disk image, they can trust systems booted from file systems derived from a netROTI installation. By configuring their hardened images and deploying them with netROTI, they can install all of their machines automatically and receive a proof from each machine, showing whether it was booted from a compromised file system.

We implemented a netROTI system for a Eucalyptus cloud environment.[2] It added only an 8-second fixed overhead plus 3 percent of image download time to the network installation process and automated verification for the administrator. The result is verifiable, automated network installation, even over an untrusted network.

Network Boot Installation

We begin by describing the process of network installation, followed by the possible attacks on this procedure and the security guarantees required for a trusted installation.

Current Network Installation

Organizations with large system deployments install and maintain their systems differently from typical desktop users. Individual systems are typically installed using an optical disk or USB drive, but the long installation process, physical media requirements, and specific customizations in that environment make the approach impractical for hundreds of machines. Instead, administrators use network-based installation techniques that employ customized automated installer images or disk cloning to rapidly upgrade out-of-date systems or restore compromised servers to their proper state.

From speaking with administrators in several large companies and our own university, which supplies computing resources for over 40,000 students, we found the most common disk-cloning tools to be Symantec's Norton Ghost (www.symantec.com/norton/ghost), Acronis True Image (www.acronis.com/home computing/products/trueimage/), or custom-designed tools that use a variety of free and open source utilities. Other services, such as Microsoft's Windows Deployment Services (http://msdn.microsoft.com/en-us/library/aa967394.aspx) and Rocks (www.rocksclusters.org), automate installation tasks. All these tools function by loading a client over the network at boot time, connecting to a management server, and downloading files, such as a preconfigured disk image or installer programs. This reduces deployment time and lets administrators harden a single installation and replicate it among systems that perform similar tasks, such as virtual memory managers (VMMs) in a cloud or employee workstations.

There are several methods of bootstrapping an installer client, but the Preboot Execution Environment (PXE) is among the most common.[3] Figure 1 briefly illustrates a network install using the PXE protocol. First, the system to be imaged, which we call the *client system*, boots into the PXE firmware loaded from the network interface card (NIC) firmware. Next, the client starts the protocol by (1) broadcasting a DHCPDISCOVER request on port 67 with an additional PXEClient extension tag. A Dynamic Host Configuration Protocol (DHCP) or proxy DHCP server responds with a DHCPOFFER on port 68, providing an IP address and a list of boot servers. The client then (2) sends a DHCPREQUEST to a boot server and gets a DHCPACK message with the file name of a network-boot program (NBP) that it retrieves from the boot server via Trivial File Transfer Protocol (TFTP). The client then executes the NBP, which can request additional files such as a kernel or modules required for the client's hardware.

The NBP sets up the installer client using files either downloaded from the boot server or retrieved through protocols like Network File System or HTTP. Finally, the client (3) contacts the image server and requests a disk image. After the installer has written the image to the hard drive, the client performs additional configuration steps, such as setting up the hostname and networking. Finally, the machine reboots into the newly imaged operating system (OS).

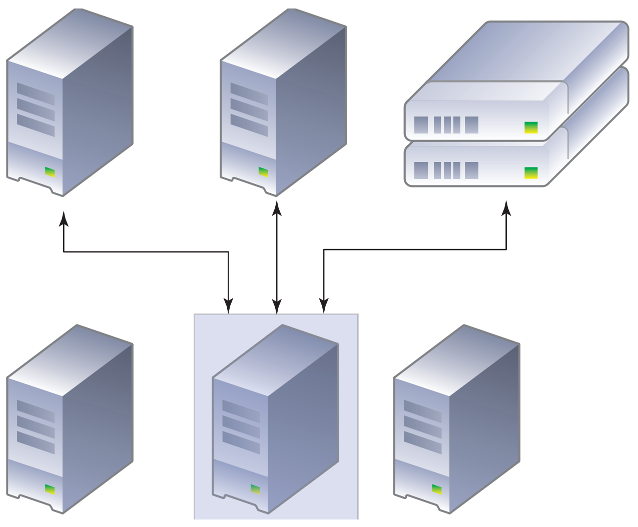
Figure 1. A network install bootstrapped via Preboot Execution Environment (PXE) boot. The client system (highlighted in blue) loads the PXE boot firmware, (1) initiates a DHCP request on the local subnet to set up basic networking and locate a boot server, then (2) requests a network boot program from the boot server and executes it. Finally, (3) the installer connects to the image server and begins transferring the disk image to the target system's hard disk, after which the system reboots.

Attacks on Network Installation

To ensure the correct operation of systems within large installations, administrators must be able to prove system installations and configurations that use high-integrity code and data. Although the techniques we've mentioned automate installation, they don't enable verification that a system booted from a properly installed file system. Potential attacks on the installation or on later system modifications could corrupt a server and lead to a host of attacks from within a large deployment like a data center.

For example, during the installation process described in Figure 1, the client system could potentially corrupt a server by loading malicious PXE firmware from the NIC installed during a previously compromised state. Another demonstrated source of corruption is a remote attack that compromises NIC firmware over the network.[4] In either case, the firmware could lead to direct attacks on the system's memory.

Another vector for attack exists when the PXE client searches for the boot server. Because the PXE client relies on information from local or proxy DHCP servers, a compromised server acting as a rogue DHCP server on the local subnet could trick the client into downloading a malicious NBP and installing a rootkit. At the network level, an attacker could modify unencrypted data sent to the client on the wire or perform a man-in-the-middle attack to tamper with the installation.

After installation, a system remains vulnerable to attacks that place rootkits on the file system or make malicious changes that persist even after a system reboots.

Securing Network Boot Installation

Securing a network installation requires showing that the installed system derives from the expected origins, installer, and disk image. Although not everyone might trust the installer or disk image, those who do would be willing to work with such a system if it could be verified. In this case, we envision data center administrators being able to leverage such trust because they specify the installer and disk images that can be loaded.

Verifying a system's installation requires accurate measurement and reporting methods. Recent work in trusted computing has examined the challenge of building trust in commodity systems. Trusted hardware, such as the Trusted Platform Module (TPM) and new extensions to Intel and AMD processors, offers various trust primitives. Using this hardware support, systems can generate attestations of a platform's critical code and data, which remote parties can verify. Bryan Parno and his colleagues survey the broad range of applications to which researchers have applied these trust primitives.[5]

However, verifying installation isn't very useful by itself because the machine will be immediately rebooted after installation and might be rebooted multiple times before any subsequent reinstallation. An administrator must therefore be able to verify that a system was booted from an expected installation. Our netROTI method lets an administrator verify that the file system at boot time is linked to the installation origins—that is, the installer and the disk image. This doesn't prevent the system from coming under runtime attacks, such as buffer overflows, but runtime attacks that modify the file system will be detected at next reboot.

Any secure network installation must be practical. A key question is whether the installed file system is sufficiently stable to enable such a verification. In an initial experiment,[6] we found that netROTI dynamically modified only three files of a privileged VM system configuration during its execution.

Our approach prohibits manual system updates because they are ad hoc. For administrators, a clean, automated install is preferable to manual modifications, anyway. Administrators that wish to push incremental updates to systems will need to extend the install-time proofs to cover these modifications to the file system.

The netROTI Method

The netROTI method links the installed system verifiably to a particular source.

Trust and Threat Models

For our design, we assume a trust model in which the physical hardware is safe from attack and is implemented correctly. We also assume that an administrator or software provider exists who has the authority to deem particular code and data, such as the installer and disk image, as trustworthy. We don't assume that such trust is placed correctly; our goal is to prove that a system links to a particular origin certified by one or more authorities. Thus, an administrator can trust in a system based on the administrator's trust in authorities' ability to certify systems. We also assume trust in the data center administrators and hence do not address insider attacks. Finally, we don't consider attacks on the cryptographic algorithms used nor attacks on the PKI or authentication procedures for establishing identities, such as direct anonymous attestation.[7]

For our threat model, we consider an attacker that can modify or inject data on the network, impersonate various services, and compromise other hosts on the network. With these attacks, the client could load a malicious installer that could, in turn, install a vulnerable or malicious disk image or otherwise compromise device firmware. An attacker could also change the contents of the client's disk after installation and perform attacks on the running system.

Reporting attacks on the system's runtime state is outside the scope of the work we report here, but the netROTI does provide a root of trust for detecting these security violations by giving a proof of the system's initial integrity at boot time.

Installation Phases

The netROTI installation method cryptographically links the installed file system with the installer and source used in the installation. Figure 2 illustrates six installation procedure phases. Each phase has a specific goal and associated tasks to achieve it.

Preinstall phase. The first phase is a manual step in which the administrator prepares the client system for installation. The phase tasks configure components that enable generation of ROTI proofs. This phase is trusted axiomatically because the administrator performs the tasks manually.

To prepare the system, the administrator configures the client BIOS to boot from the network and installs the client's Root of Trust for Measurement (RTM)

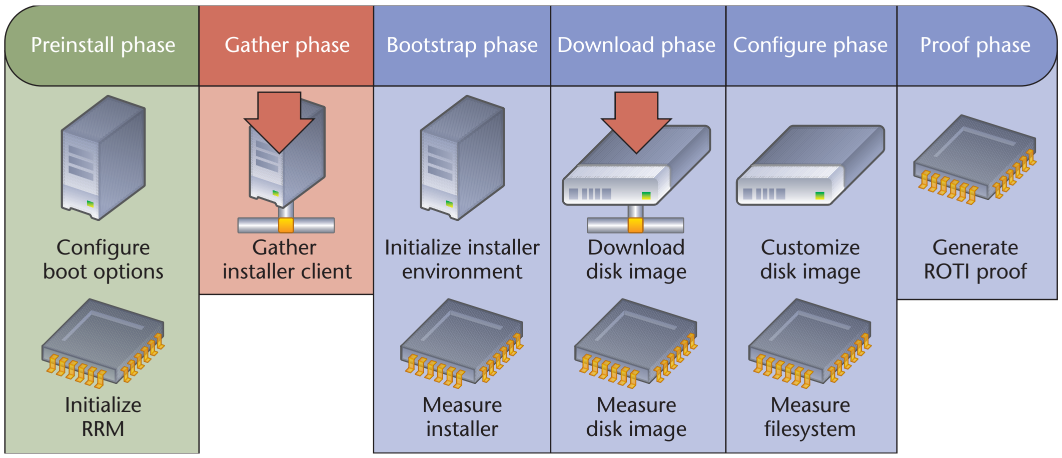

Figure 2. The netROTI installation process timeline. The administrator configures the client in the preinstall phase. Next, the client gathers the necessary files to install from the network; the phase is shaded red because it is unmeasured and need not be run by trusted code. The client then downloads the installer and bootstraps a secure environment, which measures the installer. Next, the installer downloads, measures, and configures a disk image to place on the local disk. The resulting file system is then measured by the installer and, in the proof phase, it generates a proof of the system's root of trust for installation—that is, a ROTI proof. The blue phases measure the installer code and data before executing them.

with keys that identify it uniquely. The installation process uses the RTM to record and report critical code and data.

Gather phase. The goal of the second phase is to retrieve the installation image and installer. The client gathers the necessary files to install from the network. Figure 2 highlights this phase in red because it might be performed by untrustworthy code. However, we need not measure this phase because we start the install in the bootstrap phase from a known state, using only these inputs.

The client machine first loads the network boot firmware to obtain network access and locate the boot server. Then it retrieves an NBP that downloads the additional installer files, the installer kernel, a RAM disk containing the installer code, and a bootstrap program that sets up a secure environment for the installer.

Bootstrap phase. The client then downloads the installer in the bootstrap phase, which is the first of three phases linking the installer and image to the client's file system for building a ROTI proof in the final phase.

Because the gather phase performed unmeasured operations, it opens the possibility for malicious code to have been loaded into memory. Therefore, the bootstrap phase initializes a secure execution environment for the installer, establishing a clean starting point for measuring subsequent installation process operations. A CPU-supported technique, called *late launch*, achieves this goal by taking a piece of code, recording it in the RTM, and effectively rebooting the system before executing the code in a region of protected memory. This memory protection prevents attacks both from potentially malicious resident code loaded before the installer and from external devices that have direct access to memory.

Once the installer kernel is launched, it measures the installer's RAM disk, unpacks it into memory, and begins the next phase.

Download phase. After the installer is initialized, it enters the download phase. The goal is to retrieve and measure the disk image before installing it. First, the installer prepares the local system's basic networking and partition table to enable retrieval and writing of a disk image to a clean disk. The installer also measures the disk image and records it in the RTM so that a verifier can later identify the trustworthiness of the downloaded disk image. This task helps detect attacks on the disk image while it's in transit and malicious images from compromised or rogue image servers.

Configure phase. Next, the installer configures the downloaded disk image to the target system. This includes setting up networking, file-system tables, devices, security policies, Secure Shell (SSH) host keys, and so on. The installer also generates signing keys used by the RTM for generating attestations and the ROTI proof.

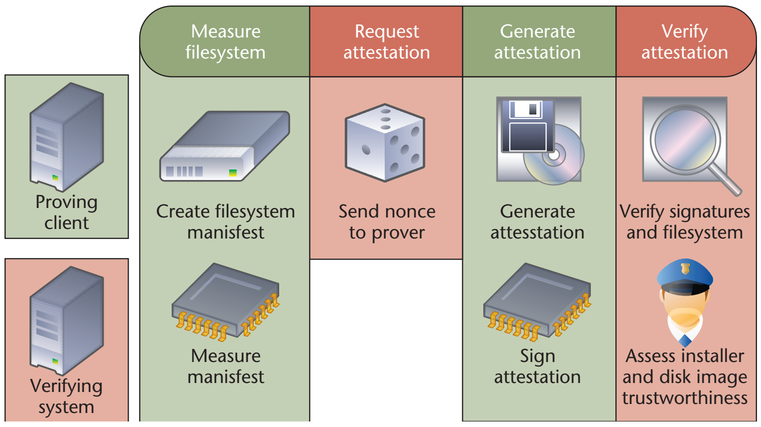

Figure 3. Timeline of the netROTI validation process. The client measures its root file system and generates an attestation of the installed and current file systems, which the verifying system can check.

Next, the installer modifies the system's startup scripts to measure the root file system at boot time. The resulting file system manifest is included in attestations so the verifier can inspect any modifications to the file system.

Proof phase. In the last phase, the RTM generates a ROTI proof that ties the final installed file system to the installer and disk image for runtime verification that the client derives from these inputs.

The ROTI proof is a signed tuple, $R = \text{Sign}(F, D, I)_{K^-}$, where K^- is a private key that identifies the physical machine and is endorsed by its RTM. This tuple acts as a proof, showing that a machine possessing K^- was specifically configured by installer I using disk image D to produce file system F. A verifier can inspect the ROTI proof to determine whether it believes the combination of I and D results in a trustworthy installation (that is, F). Upon completion of the ROTI proof, the system reboots into its newly installed file system.

Verification

Once the installation procedure is complete, the system is ready for verification, proving that the current file system used a particular installer and disk image. The verifier can then make a trust decision whether the combination of installer and disk image came from a source they trust.

Figure 3 illustrates the validation procedure. During boot, the client system loads the initial RAM disk (initrd) containing a program to measure the root file system and generate a manifest of hashes for each file. Once the root file system is mounted, the system starts a network-facing attestation daemon to handle attestation requests. When a remote verifier wishes to inspect the proving system, it sends a nonce to the attestation daemon, which builds an attestation and returns it to the verifier. The attestation is a signed statement $A = \text{Sign}(R, F', N)_{K^-}$, where R is the ROTI proof, F' is the current file-system manifest, N is the nonce, and K^- is the system's private key endorsed by its RTM.

The verifier checks that the signatures on both A and R are from keys endorsed by the proving system's RTM. Verifying the identities of keys is outside the scope of the work we report here, but we assume that the administrator deploying the systems maintains a PKI. Once the verifier establishes the identities of the signing keys, it assesses whether it trusts the installer and disk image in R to produce a trustworthy installation. If it does, it then compares F' and F in R to see how they differ. If no security-critical files have changed, the verifier can assume the current system booted into a file system that was produced by a trusted installer and disk image.

Implementing netROTI

Before describing our proof-of-concept netROTI implementation, we introduce some background on the trusted computing components we use.

Trusted Computing Primitives

Two key netROTI mechanisms are TPM attestations and the new dynamic root of trust.

The TPM is a secure coprocessor, attached to the motherboard. It provides several features, including nonvolatile RAM (NVRAM) for storing cryptographic secrets and a set of platform configuration registers (PCRs) for storing arbitrary data measurements. The TPM also contains a public key pair, the endorsement key (EK), which uniquely identifies the TPM device and associated client. Using the EK, the TPM can generate and certify other keys.

The TPM's main runtime mechanisms are

- *TPM Extend*, which adds a measurement to a PCR by hashing the measurement value with the current PCR value to form a hash chain, and
- *TPM Quote*, which produces a signed statement over a specified set of PCRs and a nonce from a second party. This quote lets a remote verifier examine the system's state (represented by the PCR measurements) at the time the nonce is generated.

The TPM uses a signing key called the attestation identity key (AIK), derived from the EK to sign the quote. The AIK effectively identifies the quote as coming from the platform containing the TPM. A *TPM attestation* is a quote combined with the list of measurements associated with the PCRs' current state. To verify an attestation, a remote verifier validates that the measurements correspond to an acceptable operation (for example, a trusted installer load) and that the series of measurements results in the provided PCR values.

Producing meaningful attestations requires establishing a *dynamic root of trust* guaranteeing that a trustworthy entity on the system takes the measurements. A computer boots into what is called a static root of trust for measurement (SRTM) because the normal boot process is largely fixed. Normally, the BIOS takes a measurement of its firmware and option ROM in any peripheral before passing control to the bootloader, which is expected to continue the measurement chain by measuring itself, the kernel, and so on. This chain, rooted in the normal boot procedure, gives a verifier a view of how the system booted. However, there are several known attacks on the SRTM, such as TPM reset attacks, which enable an attacker to reset a PCR and insert false measurements.

To address these issues, new secure virtualization architectures, such as AMD's Secure Virtual Machine (SVM)[8] and Intel's Trusted Execution Technology, let a machine form a dynamic root of trust for measurement (DRTM) by effectively rebooting the system and executing a piece of code called a secure loader in a memory-protected region that is safe from code loaded before the loader was executed. The CPU first sets a special group of DRTM PCRs to a specific value that only the CPU can set and then measures the loader before it starts. This prevents malicious code from imitating the DRTM process by inserting secure loader's code measurement into the PCR. The DRTM is useful for bootstrapping security-critical code, such as a VMM kernel, when a system's security is difficult to assess at boot time.

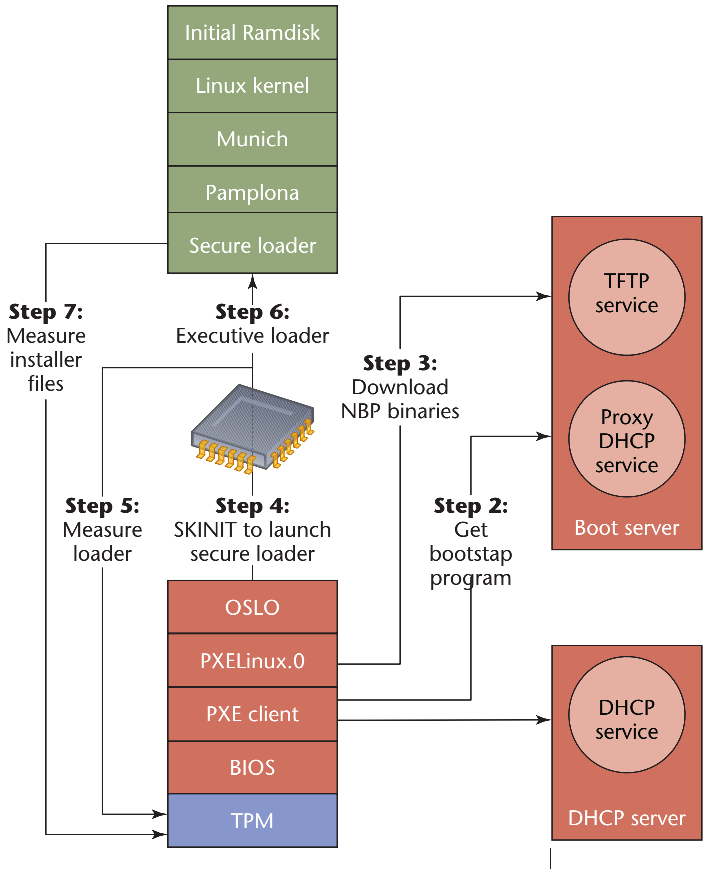

Figure 4. Proof-of-concept netROTI system. After the client follows the PXE protocol and contacts the boot server, it downloads and executes Oslo. This calls the skinit instruction to measure the secure loader block (SLB), which contains Oslo's helper binaries, Pamplona and Munich, which set up the protected installer environment. The installer kernel and initrd are then executed.

Proof-of-Concept Installation

We created our netROTI proof-of-concept implementation as a series of scripts that automates the installation process. The scripts are packed into an 11-Mbyte ext2 (second external) RAM disk that is downloaded along with a modified Linux 2.6.18 kernel and the Oslo (open source loader) bootloader.[9] Oslo is a specialized bootloader that implements the DRTM functionality in AMD processors. We use Oslo to launch the installer kernel in a secure environment.

Before installation begins, the administrator configures the BIOS to boot from the PXE firmware. The administrator must also clear the TPM of any previous administrative passwords and keys so that the installer can create its own. This corresponds to our design's preinstall phase.

In the gather phase, the firmware initiates the PXE protocol to obtain the boot server's location. The client then downloads the `pxelinux.0` NBP via the TFTP, which automatically retrieves the Oslo bootloader, Linux kernel, and installer RAM disk.

The system then enters the bootstrap phase (see Figure 4). First, the NBP constructs a multiboot header indicating the installer file addresses in memory and then executes the Oslo bootloader. Oslo consists of three executable and linkable format (ELF) binaries that perform separate stages of the DRTM process. The first binary prepares the system for the DRTM process by shutting down all but the primary CPU core and loads the second-stage binary into the secure loader block (SLB). The AMD DRTM instruction, skinit, is invoked with the SLB's entry point address as its only argument. The CPU then sets the DRTM PCRs in the TPM to 0, sets the device exclusion vector (DEV) to enable memory protection for the SLB, and sends a measurement of the SLB to the TPM. Finally, the CPU jumps to the SLB entry point that measures the installer Linux kernel, RAM disk, and boot parameters in the multiboot header, restarts the

Table 1. Breakdown of the installation time averaged over 10 installations of a Eucalyptus cloud node.

Type	Operation	Time (seconds)
Install	Download and write disk image	64.000
Install	Configuration	18.644
	Subtotal	82.644
netROT	netROTI configuration	6.740
netROTI	Measure image	1.900
netROTI	Generate TPM quote	0.890
netROTI	Measure modified files	0.390
	Subtotal	9.920
Optional	TPM setup	45.400
Optional	Generate AIK	11.220
	Subtotal	56.620
	Total install	149.184

other CPU cores, and disables the DEV protection. Finally, Oslo launches the third-stage binary that imitates a normal grand unified bootloader (GRUB) and launches the Linux kernel.

Once the installer has been bootstrapped, the installer kernel unpacks the RAM disk into memory and executes the installation script. This sets up basic device support, such as the console and networking, and starts the download phase by running a `partimage` client. This contacts a preconfigured `partimaged` server, verifies its Secure Sockets Layer (SSL) certificate against the Certificate Authority (CA) certificate in the RAM disk, establishes an SSL connection, and downloads the disk image. The installer then measures the image and writes it to the hard disk.

In the configure phase, the installer scripts configure machine-specific files, including updating devices for new hardware, networking configurations, firewall rules, and file system table entries; creating a swap partition, regenerating SSH host keys, and so on. The installer then sets new administrative credentials in the TPM and generates a fresh AIK. The TPM endorses the AIK by creating a certificate that signs the AIK's public key with the TPM's EK. The installer also installs a simple network-facing python service we wrote that acts as the attestation daemon, which services requests for attestations. The initial RAM disk is modified by the installer to generate a manifest of the file system at every boot. This manifest contains a hash of every file and is included with future attestations.

In the proof phase, the installer takes a Secure Hash Algorithm-1 (SHA-1) hash of each file on disk that has changed or was added since writing the disk image. The final step generates the ROTI proof by producing a TPM quote with PCRs containing every measurement taken during the installation process. This quote is signed with the newly created AIK from the configure phase. We use a hash of the system's hostname as the nonce because we are not concerned with the quote's freshness and care only that the ROTI proof correctly identifies the installer and disk image for this system. We tar and gzip the quote with the file manifest and list of measurements taken during installation to create the final ROTI proof file.

Proof-of-Concept Verification

Before the verification protocol between a system installed by netROTI (the *proving system*) and a remote verifier begins, the proving system boots into its initial RAM disk and then executes the measurement script inserted during installation. This script generates a manifest of the entire file system with corresponding hashes for each file. The system resumes the boot process and starts the attestation daemon. When a verifier sends a nonce to the daemon, the daemon takes a SHA-1 hash of the nonce and requests a TPM quote signed by the TPM's AIK. The quote's PCRs contain a hash of the file-system manifest taken at boot time and the nonce. The quote is then returned to the verifier with the ROTI proof file (corresponding to R in the attestation A (described earlier), the file-system manifests taken at boot (F') and during installation (F), and the AIK's certificate (the verifier has the nonce N already).

Upon receiving the attestation, the verifier first validates the quote and ROTI proof signatures. It then checks that the AIK's certificate is signed by the expected TPM's EK. Next, the verifier assesses the trustworthiness of the installer and disk image by extracting them from the ROTI proof and matching them against a list of acceptable measurements. If these are found to be trustworthy, the verifier compares the file-system manifests to see if any files have changed since installation. If no security-critical files are modified, the verifier accepts the proving system as having booted into a file system installed by a trusted installer and disk image.

Proof-of-Concept Evaluation

We used the proof-of-concept netROTI system to assess its overall impact network installation and how it addresses attacks on the installation process.

Performance

To evaluate the netROTI's overhead on installation, we performed 10 installations of a Eucalyptus cloud node's disk image across 10 systems. We created an image to be installed by manually configuring an Ubuntu server cloud in our Eucalyptus on a Dell PowerEdge M605 blade with 8-core, 2.3-GHz Opteron CPUs and 16 Gbytes of RAM on a quiescent

Table 2. A comparison of several mechanisms' ability to detect or prevent several classes of attack on an installation.

Attack Type	OSLO	Tripwire	Bitlocker	netROTI
Rootkits before install	Yes	No	No	Yes
Malicious installer code	Yes	No	No	Yes
Malicious disk image	No	No	No	Yes
Modified data after install	No	Yes	Partial	Yes
Runtime attacks	No	No	No	No

gigabit network. We then created a 387-Mbyte gzipped disk image of the 1.3-Gbyte file system.

Table 1 shows the times for each operation performed during installation. Normal installation took 82.644 seconds or 55.34 percent of the overall time. The disk-image-related operations (such as downloading, writing, and measuring the image) are a function of the disk image's size, which can be improved through more efficient compression algorithms. In particular, we found our hardware could perform SHA-1 hashes at 132 Mbytes per second, which resulted in a 1.9-second disk-image measurement time.

TPM-related operations are inherently slow due to the TPM's bus speed (33 MHz) and its low-power design. Although netROTI-specific operations added time to the install, two operations—namely, generating a new AIK and TPM setup—account for 37.95 percent of that overhead. We note that these steps create keys that could be reused across multiple installations as long as the encrypted public portions of the AIK and SRK (created in the TPM setup operation) are retained during reinstallation. Thus, an administrator could copy those encrypted files and redistribute them in the installer or have the installer copy them from the local disk before overwriting it.

Ultimately, we find the netROTI overhead to be a fixed cost of about 8 seconds plus about a 3 percent overhead for measuring the image when the optional TPM setup and AIK creation steps are reused from previous installations.

Security Evaluation

Table 2 lists a comparison of several security mechanisms and their ability to handle a range of attacks on the network installation process.

In addition to our netROTI design, we consider the Oslo bootloader alone, the file-system auditing tool Tripwire,[10] and the Windows Bitlocker file-encryption scheme.[11] Oslo uses the DRTM process both to measure malicious installer code and to defeat rootkits the system might boot into before installation, but it cannot address any other attacks. Tripwire is an auditing tool that creates a digitally signed log of the installed file system that administrators can query to detect changes. This prevents attacks that change the disk contents after installation, but it cannot guarantee anything about the file system during installation. Bitlocker encrypts the file system and optionally uses the TPM to verify that the early boot phase has not been modified before decrypting the disk. This prevents offline attacks but not modifications to the disk after decryption. The netROTI uses Oslo for protection against rootkits and to record malicious installers. It also measures disk images before installation and uses the ROTI proof combined with boot-time file-system measurements to detect changes. However, none of these approaches directly address attacks on the installed system at runtime.

The key advantage of the netROTI over these other approaches is its ability to provide an attestation of not only the file system but also the installation environment that produced it. The other solutions in our comparison prevent attacks at various stages of the installation process, but none of them can speak for the trustworthiness of the installer that produced them. By using secure hardware to measure before using each critical component during installation, netROTI creates a verifiable proof of the file system's origin.

Our evaluation demonstrated the netROTI protects against a variety of attacks on the installation process and introduces only minimal overhead when optimizations are taken into account. Using the netROTI approach, administrators can deploy systems via network installation and verify those systems have booted into a file system produced by the desired sources. □

Acknowledgments

This work was supported by NSF grants CNS-0931914 and CNS-0627551. We also thank Luke St. Clair for his efforts in designing Root-of-Trust Installer, Erik Steigler and Mark Tamminga for their insights into network installation, and Bernhard Kauer for his design of and assistance using Oslo.

References

1. R.P. Gallagher, "A Guide to Understanding Trusted

Distribution in Trusted Systems," tech. guidelines NCSC-TG-008, Nat'l Computer Security Center, 1988; www.fas.org/irp/nsa/rainbow/tg008.htm.

2. D. Nurmi et al., "The Eucalyptus Open-Source Cloud-Computing System," *Proc. 9th Int'l Symp. Cluster Computing and the Grid*, IEEE CS Press, 2009, pp. 124–131.
3. *Preboot Execution Environment (PXE) Specification* (version 2.1), Intel, 20 Sept. 1999; www.intel.com/design/archives/wfm/downloads/pxespec.htm.
4. A. Triulzi, "The Jedi Packet Trick takes over the Deathstar," slide presentation, CanSecWest 2010; www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-CANSEC10-Project-Maux-III.pdf.
5. B. Parno, J.M. McCune, and A. Perrig, "Bootstrapping Trust in Commodity Computers," *Proc. IEEE 31st Symp. on Security and Privacy* (S&P 2010), IEEE CS Press, 2010, pp. 414–429.
6. L.S. Clair et al., "Establishing and Sustaining System Integrity via Root of Trust Installation," *Proc. 2nd Ann. Conf. Computer Security Applications*, ACM Press, 2007, pp. 19–29.
7. E. Brickell, J. Camenisch, and L. Chen, "Direct Anonymous Attestation," *Proc. 11th Conf. Computer and Communications Security*, ACM Press, 2004, pp. 132–145.
8. Processor-based virtualization, amd64 style. http://developer.amd.com/documentation/articles/pages/630200615.aspx.
9. B. Kauer, "Oslo: Improving the Security of Trusted Computing," *Proc. 16th Usenix Security Symp.* Usenix Assoc., 2007, pp. 1–9.
10. 10. G.H. Kim and E.H. Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker," *Proc.* 2nd *Conf. Computer and Communications Security*, ACM Press, 1994, pp. 18–29.
11. 11. *BitLocker Drive Encryption Technical Overview*, Microsoft, 2009; http://technet.microsoft.com/en-us/library/cc732774(WS.10).aspx.

***Joshua Schiffman** is a PhD candidate in Pennsylvania State University's Computer Science and Engineering Department and the lead graduate student of the Systems and Internet Infrastructure Laboratory. His research interests include operating systems security, virtual machine security, trusted computing, and cloud computing security. Schiffman has an MS in computer science and engineering from Pennsylvania State University. Contact him at jschiffman@cse.psu.edu.*

***Thomas Moyer** is a PhD candidate in Pennsylvania State University's Computer Science and Engineering Department and a researcher in the Systems and Internet Infrastructure Laboratory. His research interests include web security, systems security, cloud computing security, and large-scale network configuration. Moyer has an MS in computer science and engineering from Pennsylvania State University. Contact him at //email//.*

***Trent Jaeger** is an associate professor in Pennsylvania State University's Computer Science and Engineering Department and co-director of the Systems and Internet Infrastructure Security Lab. His research interests include operating systems security, access control, and source code and policy analysis tools. Jaeger has and a PhD in computer science and engineering from the University of Michigan, Ann Arbor. He's the author of Operating Systems Security (Morgan & Claypool, 2008) and an associate editor with ACM Transactions on Internet Technology. Contact him at //email//.*

***Patrick McDaniel** is an associate professor in Pennsylvania State University's Department of Computer Science and Engineering and co-director of the Systems and Internet Infrastructure Security Laboratory. His research interests include systems and network security, telecommunications security, and security policy. McDaniel has a PhD in computer science from the University of Michigan. Contact him at //email//.*