Leveraging Trusted Computing and Model Checking to Build Dependable Virtual Machines

Nuno Santos[†], *Nuno P. Lopes*[†‡]
[†]*INESC-ID / Instituto Superior Técnico, Universidade de Lisboa*
[‡]*Microsoft Research*

Abstract

In the last years, it has emerged a market of *virtual appliances*, i.e., virtual machine images specifically configured to provide a given service (e.g., web hosting). The virtual appliance model greatly reduces the burden of configuring virtual machines from scratch. However, the current model involves risks: security threats, misconfigurations, privacy loss, etc. In this paper, we propose an approach to build dependable virtual machines. It is based on trusted computing and model checking: trusted computing allows for low-level attestation of the software of a virtual appliance, and model checking provides for the automatic verification of the software's high-level configuration properties. We present our approach, and discuss open research challenges.

1 Introduction

Fueled by the popularity of virtualized platforms, in particular cloud computing, a marketplace of virtual appliances has grown. Intuitively, a *virtual appliance* (VA) is a *virtual machine image* (VMI) that behaves like a single program in the sense that it is tailor-made for a specific service (e.g., web or database hosting). Typically, a VA is built by a party (the *creator*) who installs and configures the operating system and necessary software packages. The creator then uploads the resulting VA image to an online repository, such as Amazon EC2's [1], making it available to others. From a given VA image, a *user* can bootstrap VM instances where data can be processed. Such a model is very convenient for users, because it relieves them from the error-prone and burdensome task of manually assembling the VMI. Creators also have incentives, such as monetary retributions, publicity of their software, publicity of the infrastructure, or simple common good. Incentives from both parties help explain the proliferation of VA images (as of May 2011, [23] reported 8448 Linux and 1202 Windows VMIs in Amazon's datacenters) and the diversification of VA related services [2, 4, 5, 8], in the cloud and beyond [1, 6], namely in enterprises [9], universities [7, 10], and public testbeds [16].

However, the current virtual appliance model offers scant dependability assurances to users and creators. Independent studies [13, 15, 23] analyzed thousands of Amazon EC2 images and found numerous images containing: malware, obsolete and unpatched software, unlicensed software, and poorly configured images (e.g., with partially installed packages, or corrupted configuration files). Users of such images could have their data processed in unexpected ways or incur serious security risks. Same studies report risks to VA creators too. In fact, by analyzing existing and deleted files from VMIs (e.g., including logs), researchers found credentials and key material (e.g., in SSH files, shell logs), passwords (e.g., for MySQL databases), users' browsing history, personal files, IP addresses of the creator's machines, etc. Such sensitive data was unconsciously left over in many VMIs by their respective creators and could be abused by malicious users.

In this paper, we make the case for a virtual appliance model that provides stronger dependability assurances to both creators and users. In our model, both parties can specify the dependability properties they wish a VA to satisfy and a system automatically verifies whether such properties hold before uploading a VA to the repository (in the producer's case) or before using the VA right after its VM instantiation (in the consumer's case). By this, producers could specify privacy restrictions to make sure their VAs do not leak sensitive secrets, and consumers specify correctness and security properties for their VAs. To make this possible, we propose an approach based on trusted computing and model checking techniques in which the actual configuration of a VA is generated from an abstract model. Trusted computing guarantees that a VA is bound to a given model, and model checking allows for the verification of the dependabilities properties of users and creators against the abstract VA model. Next, we provide some background, and then describe our approach and discuss hard technical challenges that still need to be overcome.

2 Background and Related Work

The current virtual appliance model. A *virtual appliance* (VA) is a virtual machine image (VMI) especially configured for providing a well-defined service, and it is targeted to run on a specific VM monitor, such as Xen [11]. Table 1 lists a few popular VAs and their respective configurations. LAMP, for example, is a widely

Appliance	Type	Description	Software Setup
LAMP	WH	Web platform commonly used to run dynamic web sites and servers	Linux, Apache HTTP Server, MySQL, PHP
Email Server	IT	Email server with SPAM and virus filtering, and IMAP access for clients	Linux, Postfix, SpamAssassin, ClamAV, Dovecot
MS SQL	IT/WH	Relational database server	Windows, MS SQL Server
File Server	IT	Network attached storage service (supports SMB, SFTP and rsync)	Linux, Samba, vsftpd, rsync
Joomla	WA	Content management system	Linux, Apache HTTP Server, MySQL, PHP, Joomla
Hadoop	DC	Open source map-reduce framework	Linux, JRE, Hadoop

Table 1: Examples of popular virtual appliances. Columns indicate the name, type, description, and software of VAs. The type can be: web application (WA), web hosting (WH), IT infrastructure (IT), and distributed computing (DC).

used VA that provides web hosting services. It is built out of four software components: Linux, Apache HTTP Server, MySQL, and PHP. To create an appliance such as LAMP, the creator uses a monitor-specific toolchain to perform the following sequence of steps: (i) create a clean slate virtual machine, (ii) select a target OS (e.g., Linux) and bootstrap the newly created VM to the preselected OS, (iii) install and configure the OS, and (iv) install and configure the remaining software packages (e.g., Apache, MySQL, and PHP), and (v) shutdown the VM. The toolchain generates a large file containing the data and meta-data of the VA image, which can then be uploaded to a repository and be instatiated by authorized users to VMs on a platform such as Amazon EC2 [1].

Dependability of existing virtual appliances. Although generating VAs is rather straightforward, ensuring they work correctly and securely is hard. This is because of their complexity. A VA requires a guest OS and numerous other packages containing system tools, libraries, and applications (see Table 1). This software depends on configuration settings that must be properly set up, e.g., by creating login or database credentials, defining user and process permissions, and applying security patches. Vulnerabilities can be easily introduced given the heterogeneity and number of configuration files to be updated, allied to the fact that this operation is performed manually and in an ad-hoc fashion. For users, misconfigurations (e.g., using unpatched software version or file permissions improperly set) can lead to security breaches or incorrect behavior. For creators, privacy breaches can arise from sensitive data left over in the VA. Such vulnerabilities have been previously found in numerous public VMIs [13, 15, 23].

Risk mitigation in commercial cloud services. Currently, commercial cloud services provide no mechanisms to mitigate such risks. At best, cloud providers recommend their customers to follow a list of best practices when producing and sharing VAs [3]: "build AMIs using the most up-to-date operating systems, packages, and software", "architect your AMI to deploy as a minimum installation to reduce the attack surface", "be aware of the top 10 vulnerabilities for web applications and build your applications accordingly", etc. However, neither users nor creators of VAs can be really sure that these recommendations have been followed and whether such guidelines are sufficient.

Risk mitigation based on VMI sanitization. Prior research has focused on reducing the privacy and security risks incurred by VA creators and users. In particular, Wei et al. [36] proposed a management system that provides high integrity of VMIs and enables their secure sharing. Deployed in the back-end of cloud services, this system automatically runs programs (called filters) to scrub creators' data from VMI images and sanitize them (e.g., by removing malware, detecting unlicensed software, upgrading obsolete software, and patching insecure software); this procedure gives users better assurances of the VMI's software quality. However, because it relies on heuristics, sanitization can only be partially guaranteed. Moreover, it is silent with respect to conveying to customers what software is shipped in the VMI and whether is has been correctly configured.

Risk mitigation leveraging trusted computing. Another class of systems leverages trusted computing hardware to attest the software state of VM instances. Existing systems, however, represent such a state either partially or too low level to be meaningful for creators and users. In the simplest form of attestation [14, 17, 33], the state representation of a VM instance consists of an authenticated hash chain of the software stack's bootstrap sequence. Every time a machine boots, the hashes of the software are computed and stored in a set of registers located in a trusted piece of hardware, typically the Trusted Platform Module (TPM) [19]. These hashes can be securely transmitted to a user by signing both the hashes and a nonce previously chosen by the user using a private part of an asymmetric key pair called Attestation Identity Key (AIK). Because the private part of the

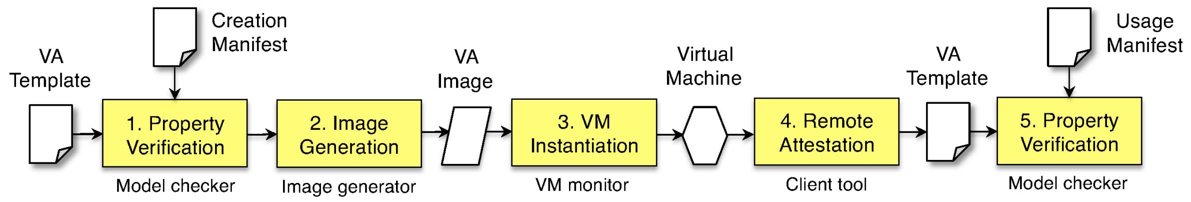

Figure 1: Workflow of the depliance model.

AIK is accessible in its cleartext form inside the TPM only, the user can authenticate the hash chain by validating the signature against the AIK's public key. These hashes, however, are too coarse grained; they typically cover measurements of the entire OS [17] or of simple functions only [29], lacking extensive coverage of software packages and their configurations. More advanced attestation schemes [20, 34] raise the level of abstraction, but focus on single processes only. Other proposals broaden the scope of attestation to cover system wide properties. Jaeger et al. [24] suggests that the attestation report includes a mandatory access control policy enforced by the OS. While this method enables remote users to check system-wide integrity policies, it does not convey information on how the system or applications are configured. In an alternative scheme [31] developed for Linux, the attestation report includes integrity measurements of programs (and optionally configuration files) that are allowed to be executed in the system. With this scheme, individual programs and respective configuration files can be validated before they get executed. However, to validate the semantics of configuration files, users need to implement specific validation scripts which is a burdensome and error-prone task we wish to avoid. Property-based attestation [30,32,33] allows for mapping low-level hash-chain values to high-level human-readable attributes. For example, for a LAMP VMI, its hash chain would be mapped to string "LAMP" and the attestation protocol returns this string signed by a trusted party that endorses the correctness of the mapping. In such systems, however, endorsement of property mappings is performed manually by humans, and therefore subject to the errors we seek to mitigate.

Verification of configuration properties. Existing work focuses primarily on detecting misconfigurations of deployed systems [35, 37] or on reconfiguring them automatically [12]. In our work, we validate the configuration of VMIs prior to their deployment. For this purpose, we adopt model checking, a technique that has been successfully applied to the verification of programs [25], but not of software configurations.

3 Proposed Approach

To improve dependability of virtual appliances, we propose a model for building virtual appliances such that creators and users have the ability to check configuration properties of a VA before its effective creation and usage. Our model builds on prior work on trusted computing and adopts model checking principles to validate high-level configuration properties of virtual appliances. A VA built under our model is called *depliance*.

Overview. Figure 1 represents the workflow of a depliance's lifecycle. Each step is assisted by dedicated tools or runtime components. The creator starts the entire process by producing a formal model of the VA (*template*) and defining a specification of configuration properties that must be satisfied (*creation manifest*). Next, the creator runs a model checker to verify whether the template satisfies the required dependability properties (1). If so, the VA image can be generated safely using a generator tool (2). Later, based on the VA image, a user can instantiate a VM on a hosting platform using a platform-specific client tool (3). After bootstrapping, the VM can be remotely attested by the user. Attestation yields the template that originally produced the image of the running VM (4). If attestation succeeds, the user can determine whether the VM is configured according to her needs by specifying relevant properties in a usage manifest file, and verifying, using a model checker, that the template satisfies such properties (5). If validation passes, the VM is deemed trustworthy, and it is ready to be used. Essentially, compliance guarantees are given to the creator and users in steps 1 and 5, respectively. Next, we describe this workflow in more detail using a simple depliance example.

Depliance modeling. To build a depliance, the creator must first write a template in a domain-specific language (DSL). The template specifies a model of the VA configuration that describes all *modules* necessary and sufficient for building the VA. To illustrate this concept with an example, Figure 2 represents a simplified template of a Joomla depliance (see Table 1). The boxes represent the modules of the Joomla depliance: virtual hardware, and the software packages Linux, MySQL, Apache HTTP Server, PHP, and Joomla. Arrows indicate dependencies between modules. Each module features a set of *configuration attributes* and *files* annotated to the module's left- and right-hand sides, respectively. Together, configuration attributes and files (which include executables, libraries, etc.) parameterize the behavior of a given module. As we explain next, from a VA template it is possible to (i) verify relevant properties, and (ii) generate the VA image.

Property verification. Based on a template, the creator (and users) can check for specific properties using a model checker. To enable property verification, the behavior of each module is expressed as a state machine. Each state machine depends on its module's configuration and file attributes. The creator specifies the verification properties in the DSL as logic conditions over such attributes, enabling a model checker to validate them (or not, if there is indeed a problem). This general technique allows for the verification of multiple properties, such as the following ones:

- **Efficiency:** For example, in the Joomla depliance, ensure proper configuration of the maximum number of concurrent threads for Apache (*MaxClients*) and PHP (*nthreads*) based on the memory (*Mem*) and cores (*CPU Cores*) available on the virtual hardware.
- **Confidentiality:** For example, validate user and password data of modules (e.g., *Root Pass* of MySQL or *Admin Pass* of Joomla) by checking that the private identity of the creator is not revealed and that the modules' default, and therefore insecure, access credentials have been overridden.
- **Integrity:** For example, validate the version of each module's software, and check for missed patches.
- **Authenticity:** For example, check that only certified software can be installed in the VA, or even restrict software installation to certain entities.
- **Accountability:** For example, provide that the VA is configured with a logging mechanism.
- **Reliability:** For instance, verify that a backup software module is properly set up.

Image generation. From a properly validated VA template, the next step is to generate a complete VA image. Since each module has specific executables and configuration files, this operation is performed by module-specific *helper programs* that translate the high-level model representation into low-level package-specific installation and configuration commands. For example, the helper program for module "Apache HTTP Server" must interpret the configuration attribute that sets the maximum number of worker threads (*MaxClients*) and edit Apache's configuration file */etc/httpd.conf* accordingly. To make sure

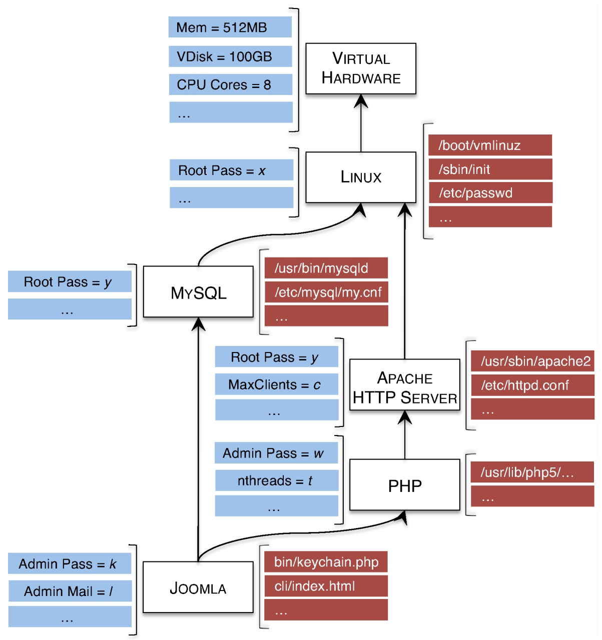

Figure 2: Simplified template of Joomla depliance.

that the VA image corresponds to the template's description, helper programs must be correctly implemented by trusted third parties, such as software package builders, open source communities, specialized certification companies, etc.

VM instantiation. From the VA image produced previously, VM instances can be bootstrapped and executed on a virtualized platform using standard client side tools.

Remote attestation. After booting up a VM, a user must check whether the VM currently instantiated satisfies her dependability requirements. To make this possible, the user's client tool implements a remote attestation protocol that yields a digitally signed copy of the original template, allowing the user to validate relevant properties using a model checker. To ensure that the template returned by the attestation protocol corresponds to the packages installed and configured during the VM generation stage, the VMI must be produced by a trusted entity that guarantees the overall integrity of the VMI and includes a cryptographic digest of the template in the VMI.

Usage scenarios. Our virtual appliance model is suitable for a variety of usage scenarios targeting the cloud setting. In the most obvious case, creators are responsible for modeling and building the VA. The cloud provider stores the VA images and manages the infrastructure where VM instances run. Users run client side tools that interact with the VM instances externally. While this model suits nicely the current cloud paradigm, the creators must be trusted to build the VMI correctly and therefore no VMI integrity assurances can be given in the face of a malicious creator (who can intentionally tamper with the VMI). To avoid trusting the creators, the VM bundling operation can be outsourced to the cloud provider: if the users trust the cloud provider for hosting VM instances, it is not unreasonable to trust the same provider to generate VA images out of templates submitted by creators. However, if no single cloud provider is involved, or if it is not trusted, the VA bundling and hosting stages can be performed entirely by the user, who only needs to obtain the VA template from the creator and run the necessary tools accordingly. Naturally, the depliance model could be employed in other contexts beyond cloud computing, e.g., within enterprises or for personal use.

4 Open Challenges and Future Work

We are currently building the framework tools for depliance implementation support. In this section, we discuss the most relevant open challenges we are facing, and how we plan to overcome them.

Comprehensive appliance modeling. In a typical VA, all software combined counts thousands of configuration options. If the DSL exposes all these options to the VA creator, templates will result in complex and overwhelming specifications. On the other hand, restricting the set of options that can be tuned at the DSL level can hinder the configuration flexibility of depliances. To combine manageability and flexibility, we borrow some ideas from Click [27], a router emulator that allows for the modular development and integration of software router components. Essentially, our approach will be to decouple the language abstractions of the DSL from the software packages' so that a configuration attribute in DSL needs not to map directly to a specific option in a configuration file, but can be made to configure a set of low-level options. This allows for different degrees of fine-tuning capability.

Efficient property verification. The properties listed in Section 3 can be verified using standard model checkers, such as HSF [18], PRISM [28], SPIN [22], and Z3 μZ [21]. However, which model checker to use depends on several factors, including the type of property. In fact, different properties vary in difficulty. Such a property diversity also makes it unclear, at this point, what will be the shape and size of the verification conditions. Therefore, further investigation is required to determine: (i) which logic to use in order to encode the verification conditions (e.g., temporal or SMT logics), and (ii) whether existing verification tools, such as model checkers, SAT/SMT solvers, and theorem provers, are able to efficiently discharge the verification conditions. New logic fragments and/or new tools might be necessary to verify depliance models. We also plan to investigate whether our technique could be employed to mitigate side-channel attacks, e.g., by detecting the presence of cryptographic libraries resilient to timing attacks.

Correct VA generation. To generate a VA from a template, we rely on trusted helper programs that translate high-level template specification into low-level commands responsible for implementing the necessary installation and configuration operations. The smaller these programs are, the higher are the chances that their code is correct and that existing software verification techniques can be used to verify these programs [26]. To make helper programs small, we will exploit existing similarities in the configuration mechanisms commonly used in software packages. Note that we assume that the programs to be installed are correct; it is not within the scope of our work to verify the correctness of application code.

Untrusted VA generation. To make sure that remote attestation is correct, the VA generation performed by helper programs cannot be adulterated, otherwise no assurances could be given to users that the VA template returned by a successfully attested VM would reflect the VA implementation. Therefore, in order to be trustworthy, the VA generation step must be performed by a trustworthy entity, namely a trusted cloud provider or the user himself. Being able to delegate the VA generation to an untrusted party, such as the VA creator, could further facilitate this operation. Achieving this, however, is hard. To overcome this challenge, we wish to explore whether it is possible to create an integrity measured log of operations performed during VA generation that could be included in the attestation report, so that users could check it before the model validation stage.

5 Conclusions

This paper proposed a novel approach for improving dependability of virtual appliances. Our approach enables building virtual appliances — referred to as *depliances* — whose configuration properties can be verified by respective creators and users. To achieve this, we leverage trusted computing and model checking techniques.

To the best of our knowledge, this is the first work to focus on verifying the configuration of full virtual appliances. In this paper, we discussed open research challenges of implementing our approach, endeavor that we leave out for future work.

Acknowledgments: We would like to thank our shepherd, Allen Clement, and the anonymous reviewers for their feedback and insight. This work was supported by national funds through FCT (Fundação para a Ciência e Tecnologia), under project SpecSTM - PTDC/EIA-EIA/122785/2010, and INESC-ID multianual funding - PEst-OE/EEI/LA0021/2013.

References

[1] Amazon EC2. http://aws.amazon.com/ec2.

[2] Amazon Machine Images (AMIs). http://aws.amazon.com/amis.

[3] AWS Marketplace: Security Guidance for AMI Developers. https://aws.amazon.com/marketplace/help/200897460.

[4] BitNami Virtual Images. http://bitnami.org/learn_more/.

[5] CUBRID Virtual Images. http://www.cubrid.org/virtual_machine_images.

[6] Microsoft Azure. http://azure.microsoft.com/.

[7] UC Berkeley - Cloud Computing Services. http://ist.berkeley.edu/services/is/virtual-servers.

[8] VMWare Solution Exchange. https://solutionexchange.vmware.com.

[9] VMware Success Stories. http://www.vmware.com/a/customers/industry/.

[10] Washington University - Home Virtual Machines. https://www.cs.washington.edu/lab/software/homeVMs.

[11] XenServer. http://www.xenserver.org.

[12] M. D. Aime, A. Lioy, and P. C. Pomi. Automatic (re)configuration of IT systems for dependability. *IEEE Transactions on Services Computing*, 4(2):110–124, 2011.

[13] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro. A security analysis of amazon's elastic compute cloud service. In *SAC*, 2012.

[14] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: virtualizing the trusted platform module. In *USENIX Security*, 2006.

[15] S. Bugiel, S. Nürnberger, T. Pöppelmann, A.-R. Sadeghi, and T. Schneider. AmazonIA: When elasticity snaps back. In *CCS*, 2011.

[16] C. Cutler, M. Hibler, E. Eide, and R. Ricci. Trusted disk loading in the Emulab network testbed. In *WCSET*, 2010.

[17] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP*, 2003.

[18] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[19] T. C. Group. TPM Main Specification Level 2 Version 1.2, Revision 130, 2006.

[20] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation - a virtual machine directed approach to trusted computing. In *VM*, 2004.

[21] K. Hoder, N. Bjørner, and L. de Moura. μZ- an efficient engine for fixed points with constraints. In *CAV*, 2011.

[22] G. J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.

[23] J. H. Huh, M. Montanari, D. Dagit, R. B. Bobba, D. W. Kim, Y. Choi, and R. Campbell. An empirical study on the software integrity of virtual appliances: Are you really getting what you paid for? In *ASIACCS*, 2013.

[24] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *SACMAT*, 2006.

[25] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, Oct. 2009.

[26] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.

[27] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[28] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, 2011.

[29] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, 2008.

[30] A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *NSPW*, 2004.

[31] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security*, 2004.

[32] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *USENIX Security*, 2012.

[33] J. Schiffman, T. Moyer, H. Vijayakumar, T. Jaeger, and P. McDaniel. Seeding clouds with trust anchors. In *WCCS*, 2010.

[34] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *SOSP*, 2011.

[35] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI*, 2004.

[36] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning. Managing security of virtual machine images in a cloud environment. In *CCSW*, 2009.

[37] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou. EnCore: Exploiting system environment and correlation information for misconfiguration detection. In *ASPLOS*, 2014.