On the Feasibility of Remote Attestation for Web Services

John Lyle and Andrew Martin
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD
{John.Lyle, Andrew.Martin}@comlab.ox.ac.uk

***Abstract*—Remote attestation is a significant part of the functionality offered by trusted computing, allowing a platform to demonstrate that it is running trustworthy software. However, this technique has been criticised as impractical, citing the management overhead of maintaining lists of acceptable software configurations. In this paper we put numbers to this problem, and argue that remote attestation may not be too fragile a technology to be used for web services.**

I. Introduction

Integrity reporting is the core mechanism proposed by the Trusted Computing Group[1] (TCG) for establishing trust in a computing platform. It works on the principle that a platform can be trusted if all the software and hardware it has run (its 'configuration') can be identified and verified by a relying party. This principle has been criticised for a number of reasons. Firstly, the vast number of available applications, operating systems and hardware drivers mean that any list of trustworthy software and hardware will need to be unmanageably large. Secondly, there are few realistic methods for establishing if any one configuration is trustworthy. These problems spell disaster for trusted computing, and may cause researchers and developers to dismiss the functionality provided.

However, little effort has been spent within the trusted computing community to support or refute these criticisms. While the general arguments are difficult to deny, an assessment of their validity in different contexts is missing. The supposed wide range of configurations may not exist in embedded systems, for example. We believe that putting numbers to this problem is essential, and the main contribution of this paper is to assess the viability of using software integrity reporting (in the form of TCG remote attestation) in a web services scenario.

The paper is structured as follows. In Section II we describe remote attestation and the TCG methods for establishing trust, as well as the many problems they face. In Section III and IV we discuss our experiment, simulating the necessary updates that would be installed by a web service platform over a two and a half year period. We then analyse the results in Section V and suggest a number of ways of reducing and simplifying the process in Section VI. Section VII covers existing work in this area and in Section VIII we conclude.

II. Background

A. Trusted Computing

Trusted computing is a paradigm developed and standardized by the Trusted Computing Group[1]. It aims to enforce trustworthy behaviour of computing platforms by identifying a complete 'chain of trust', a list of all hardware and software that has been used. If a platform owner can reliably find out exactly what software and hardware is in use, they should be able to recognise and eliminate any malware, viruses and trojans. A great deal of infrastructure is required to make this idea practical, including new hardware, modifications to applications and databases of known, trustworthy platform configurations.

The technologies proposed by the TCG are centered around the Trusted Platform Module (TPM). In a basic server implementation, the TPM is a chip connected to the CPU. It provides isolated storage of RSA keys and Platform Configuration Registers (PCRs). These PCRs can be used to hold *integrity measurements*, in the form of 20 byte SHA-1 hash values. They can only be written to in one way: through the `extend(..)` command. This appends the current register value to the supplied input, hashes it, and stores the result in the PCR. A PCR value therefore reflects a *list* of individual hashes. In order to work out what individual inputs have been added to a PCR, a separate log must be kept. When this log is replayed, by rehashing every entry in order, that final result should match the value in the PCR.

The limited functionality offered by the TPM is ideal for recording the boot process of a platform. The idea being that, starting from the bios, every piece of code to be executed is first hashed and extended ('measured') into a PCR by the preceding piece of code. This principle is known as *measure before load* and must be followed by all applications. If so, no program can be executed before being measured, and because the PCRs cannot be erased, this means that no program can conceal its execution from the TPM. The first module cannot be measured, and is referred to as the *root of trust for measurement*. A platform is said to support *authenticated boot* when it follows this process as it provides a way for users to authenticate their platform's boot sequence against reference values.

B. Remote Attestation

In order for users to assess a remote machine, the TPM has a feature called *remote attestation*, which allows a platform to report the integrity measurements collected during authenticated boot. When challenged, a trustworthy platform will use the 'TPM Quote' operation to create a signed copy of its PCR values. This is then given to the challenger for inspection, along with the measurement log.

The signed PCR values and log must be verified by the challenger before the platform can be trusted. The PCRs are signed using a private key held by the TPM, guaranteeing the key's confidentiality. This is called an Attestation Identity Key (AIK) and the public half must be certified by a third party certificate authority (a 'Privacy CA'). Full details can be found on the TCG website[1] and Sailer et al. [2] have outlined a full nonce-challenge protocol.

The software running at the platform can be identified by matching the hash values in the attestation with reference data. This requires a list of *reference integrity measurements* (RIMs) contained within a Reference Manifest Database, as described in the TCG Integrity Management Model[3]. These measurements are collected from their original source: the software and hardware manufacturers. For example, Microsoft could release RIMs containing the correct hash measurements for each file in Windows Vista. Creating and maintaining this database is a challenging task, but the next step is perhaps harder. Having identified the running software, a decision must be made on the overall platform trustworthiness. This is an open problem in trusted computing research.

C. Problems with Attestation

There are some well-known issues with TCG-described attestation. These include the disclosure of platform configuration information (*privacy*), the semantic gap between hash values and platform properties (*semantic gap*), attacks on running software (*runtime*) and the practical difficulty of maintaining a whitelist of known hash values (*whitelisting*).

The first consideration is privacy. Integrity measurement requires the challenging party to identify every piece of software executed on the remote platform. This might allow them to discriminate based on their own criteria[4], [5], requiring software from only one vendor, for example. This could work against the user's best interests. Furthermore, reporting the exact hash values potentially makes an attacker's job easier[6], as he or she will be able to quickly identify which known exploits are appropriate for the platform.

Attestation has also been criticised for reporting a platform's *execution state* rather than its *security state*[5], which many consider to be the ultimate goal. These two properties are related, but there is a significant semantic gap between them. If it is not clear that one software configuration is necessarily more secure than another, what is the purpose of the attestation? To solve this problem, and the privacy issues mentioned earlier, Sadeghi and Stüble[4] introduced 'Property-Based Attestation' (PBA). In PBA, platforms provide a list of guaranteed security properties, rather than just a binary integrity measurement. This can be implemented in a number of ways, but generally assumes that at least one party can match software identity to security properties. Presumably this would be done through testing or verification, both time consuming processes. We therefore believe that the first step in simplifying this problem is to reduce the size of a platform's software configuration.

Integrity measurement can assert the identity of software when it was originally loaded, but says nothing about the runtime state of the platform[7], [8]. In-memory attacks (such as exploiting a buffer overflow) can occur which will not be reported in an attestation, but will certainly alter the expected behaviour of the machine.

Finally, the complexity of managing a large software whitelist has frequently been cited as a major problem for attestation. England[8] claims that the 4 million windows drivers (growing at 4000 per day) makes even identifying the software running on a platform a daunting task. Other researchers have made similar points[4], [9] about the number of potential software configurations. However, there are some promising counter-examples. Sailer et al. [10] show an implemented network access control system which uses a whitelist of only 25000 entries, and is designed to handle application updates. On a larger scale, Bit9 have a 'Global Software Registry'[1] which claims to contain over 6 billion entries. These two projects imply that the management problem of maintaining a large whitelist are overstated. In the rest of this paper, we assess whether this is true for a web service.

D. Web Services

Web services are a well-established approach to creating large, component-based applications[11]. Services communicate and describe themselves using standard formats, such as SOAP[12], and have interoperability as a primary requirement. Each service offers some kind of useful functionality, and they can be combined together easily, allowing the rapid creation of new, custom applications. We chose to look at web services in this paper because of their increasing popularity and standardized behaviour.

E. Attestation for Servers

Attestation is usually considered in a client-side context, with a server challenging client platforms to prove their trustworthiness[3]. The server may be required to mediate access to an important resource or service. This is the scenario that is often assumed when discussing attestation problems. The *whitelisting* problem mentioned in Section II-C appears daunting, as the numerous clients could be running any combination of operating systems and applications and the server must be able to evaluate all of them. Any whitelist or reference database will need to contain millions of entries, and the meaning of each one will be hard to maintain.

However, the opposite scenario is sometimes overlooked. A client machine may require some assurance of a *server's* correct operation. This will be particularly true when handling

[1] http://www.bit9.com/products/gsr.php

confidential information or when performing an important task, such as payment processing. Now the server must attest to the client, who in turn has to interpret the server's integrity measurement log. When only one (or a small number) of machines is considered, this should make whitelisting more tractable. Furthermore, servers are typically only performing one task. They have a distinct role, as an email or web server, for example. These rarely have graphical user interfaces, and can be run on minimal operating systems. The total amount and size of running software is much smaller than a general-purpose client machine, which may be simultaneously used for web browsing, word processing, and email. Again, this reduces the amount of software which must be trusted and will be included in an attestation log. We believe that these points make servers a better candidate for attestation.

III. Experiment Description and Methodology

We designed an experiment to quantify the difficulty of attesting a web service, by counting how many measurements would need to be maintained in an integrity measurement database. If a large number of reference values must be stored, then this would support the argument that integrity measurement is impractical.

To attest one service, the database would be as big as the number of unique pieces of software that it runs. However, software is often updated, so our experiment had to take into account the *rate of change* of the platform. We used information from two sources, the Ubuntu Linux package repository[2] and the Sun website. A two and a half year period (June 2006 to January 2009, inclusive) was studied. The rest of this section details how we set up our web service platform, and our methodology for counting updates.

A. Platform Details

We set up an Ubuntu 6.06 Linux platform and modified it to support authenticated boot. This involved compiling a customised version of the Linux Kernel, initially version 2.6.22.1, complete with the IMA[2] patch to measure executables and kernel modules. We then modified and compiled a version of the OpenJDK, in a similar manner to the patches released by the 'Trusted Computing for the Java(tm) Platform' project[13]. We used standard versions of the Glassfish Application Server to run a simple web service which answered attestation challenges. The IAIK JTSS[3] libraries were used to communicate with the TPM from the web service. Using this software, we made attestation requests from another platform and recorded the results in a database. We chose popular web service software for these experiments. Ubuntu Linux is increasing in popularity on servers[4] and the Glassfish application server was downloaded 3.5 million times as of June 2007 [14]. This makes our test platform a reasonable case study.

We have discarded BIOS and bootloader measurements. New Intel and AMD processors support a Late Launch feature, which makes verifying these components unnecessary[15]. We have also ignored hardware measurements, as these are harder to verify and should add only a constant number to our results.

B. Counting Updates

Because of the need to recompile the Linux kernel and Java to support integrity measurement, the process of counting software updates was not always as simple as just applying the upgrades and re-attesting the system.

Our first step was to get a baseline, initial attestation of the platform. This contained a list of all executables that were run, without any user logins to the machine. We then looked at how each of the applications on this list was modified by updates. The final output was a timeline, containing all files that were changed and the date the new versions were released. We have assumed a pro-active administrator who applies all patches as soon as available, but does not upgrade the entire OS distribution.

1) Operating System: We counted every new version of the kernel that was released in the Ubuntu repositories for the 6.06 distribution. We observed that every new version had entirely new hash values for each kernel module. As a result, we calculated the total number of measurements as the initial module count, multiplied by the number of kernel updates.

2) Core Executables: Programs and libraries such as bash and glibc were updated. This was simulated by counting the number of updates released in the Ubuntu repositories, and then looking at how many measured executables would be affected, with reference to our baseline attestation.

3) Java: We handled these updates through the Sun website. We assumed new version would be installed whenever available, and that the migration from Java 5 to Java 6 would happen at the first opportunity. Because we were using a customised version of the JRE (compiled from source) to support integrity measurement, we could not simply install new versions and re-run the attestation process. Instead, we compiled a list of files that our custom JRE used, and then worked backwards to see which updates modified files on this list. We anticipate that a few libraries will have been missed in this process, but expect that number to be small. Versions 5.6 to 5.10 and 6.0 to 6.11 of the JRE were counted.

4) Glassfish: We counted the number of libraries and executables associated with Glassfish through installing and running it on our customised JRE. A simple web service was run on each version. This service had only one function: returning attestations when challenged. We then upgraded it with every core release of a new version of Glassfish, as detailed on the download page of the website[5], excluding version three, which was still in beta. Because we did not modify Glassfish in any way, it is likely that a few libraries were loaded using an unmodified classloader which we did not include. Again, however, we expect this number to be small.

[2]http://packages.ubuntu.com/

[3]http://trustedjava.sourceforge.net/index.php?item=jtss/about

[4]http://www.iaps.com/2008-server-reliability-survey.html

[5]https://glassfish.dev.java.net/public/alldownloads.html

C. Configuration Files

The measurement of executables over time does not take into account configuration files. This is an oversight, because much of a platform's behaviour can be controlled through configuration. It is likely that any standard operating system would need to attest certain settings, such as firewall rules. However, there are good reasons for not including them. Firstly, it is difficult to establish which files would be important and need attesting. Some (`/etc/motd`, for example) clearly have no relevance to the trustworthiness of the platform, but knowing which ones would require an enormous amount of time. Secondly, we could not anticipate how configuration files would need to change to reflect application updates. Generally, the *number* of files would stay the same, and the content might be added to. We therefore decided to get approximate figures by measuring the total number used, and then estimating an upper bound on how many would be relevent, ignoring change over time.

We established a total figure by augmenting the IMA patch with an extra SELinux hook - `dentry_open` - and logging every access. We then booted our system and started Glassfish. We eliminated all binary files, logs, and non-configuration related shell scripts. Unfortunately, it is possible that JAR files could contain configuration settings which we have not included. We measured the number of lines in these files using `wc -l`, with comments removed.

IV. Results

Our baseline system had 277 hash values, consisting of 17 JRE 5.7 libraries, 50 glassfish v1 libraries, 4 jTSS jar files and 53 kernel modules. The rest were standard applications and shared libraries. Between June 2006 and January 2009 (32 months), 1137 measured files were updated, approximately 35 files per month. This made a total of 1414 hash values. Apart from those already mentioned, there were 13 base packages updated, including `gzip`, `udev` and `e2fslibs`.

From the log of configuration files, we found a total of 113 were read, with 49 that were either empty or considered unambiguously unimportant. This leaves 64 that might need to be measured. The total number of lines in configuration files was 6370, with just under 5000 in the 64 files we considered important. The vast majority of these (3414) were in Glassfish XML documents. We also suspect that some of the Glassfish schema files we considered significant would never change and could be attested using a hash, making line count unimportant.

TABLE I
UPDATES APPLIED BY MONTH

Month	Java	Glassfish	Kernel	Other	Total
Jun 06	12	0	53	1	66
Jul 06	0	0	53	1	54
Aug 06	0	0	0	0	0
Sep 06	0	0	53	6	59
Oct 06	8	0	0	0	8
Nov 06	33	0	0	6	39
Dec 06	0	53	0	0	53
Jan 07	0	0	0	44	44
Feb 07	0	0	53	17	70
Mar 07	10	0	0	14	24
Apr 07	0	0	0	2	2
May 07	0	0	0	0	0
Jun 07	12	0	0	0	12
Jul 07	0	0	0	0	0
Aug 07	0	0	53	2	55
Sep 07	9	88	0	2	99
Oct 07	0	0	53	20	73
Nov 07	0	0	0	0	0
Dec 07	13	67	0	6	86
Jan 08	0	0	53	1	54
Feb 08	9	0	0	0	9
Mar 08	11	0	0	0	11
Apr 08	0	57	0	2	59
May 08	0	0	0	1	1
Jun 08	0	0	53	2	55
Jul 08	0	0	0	0	0
Aug 08	9	0	0	0	9
Sep 08	16	0	0	0	16
Oct 08	0	0	0	27	27
Nov 08	9	0	53	1	63
Dec 08	0	0	0	3	3
Jan 09	0	86	0	0	86
Total	**151**	**351**	**477**	**158**	**1137**

V. Analysis and Implications

The number of hash values recorded is not sufficient to show that attestation is feasible in this scenario. This depends on the *purpose* of the attestation, the property to which the server is trying to attest. In this section we start by looking at four general properties and analyse whether our results make them achievable or not.

For the purpose of identifying running applications and checking their integrity, our results look promising. Any database is capable of storing 1414 values, and the vast majority of hashes can be obtained from a few public repositories. The only assumption that must be made is that each step in the boot chain follows *measure before load*, not allowing any unmeasured code execution. Having identified the running applications, we can then know their vendor and patch-level, which can be useful when assessing other properties of the platform. For example, Munetoh et al. [16] use this information in combination with an online vulnerability database to calculate how many vulnerabilities a platform is known to have.

The relatively small number of possible hash values means that there is no reason for an unknown application to ever be run or attested. Challengers can therefore take the presence of an unknown hash in an attestation log extremely seriously. This makes it unlikely that a server with a malicious rootkit would be trusted by a remote user. Again, we assume that all applications support integrity measurement. We believe that these results reinforce the idea that attestation is suitable for establishing that a platform *did not*, at boot-time, have a rootkit installed.

Another property we *can* attest about our platform is that nobody has logged into a terminal, either locally or remotely. This is because various executables are run at log-in, including the `pam` security applications and (locally) `/bin/login`. Any fresh attestation of a platform that does not include these

has not *yet* been logged into. This does not, however, discount logging in the administration console on Glassfish (or through any other executable) but if no executable has been run that supports remote login, it seems possible to attest this general property. This could be useful for internal monitoring, or when trying to mitigate insider threats.

A more difficult property to establish is whether or not a platform is deemed *trustworthy*. We definitely cannot use the attestation to establish a platform's *correct* behaviour, as none of the hardware of software has been analysed for this. But will the software, to use TCG terminology[3], behave as expected? At first, we might assume that this is the case, as Linux, the JVM and Glassfish do generally work in their expected way. However, when we consider one of the criticisms of attestation - that it deals with execution state, rather than the runtime state of the platform - we run into problems. Any of the running processes may have been exploited since system boot and no longer behave in their usual manner. Unfortunately, exploits for large operating systems and applications are being discovered on a daily basis, and this makes it impossible for attestation to support any claim of trustworthiness. This is a well known criticism of common operating systems, and arguably makes any attempt at establishing a much higher level of security doomed to failure[17]. We therefore cannot establish trustworthiness because we cannot assess the security state of the server through TCG attestation.

In order to move from *identified* to *trustworthy* we need to eliminate the chance of runtime exploit. This means reducing the number and size of applications running on the server, and improving the quality of the code. In Section VI we discuss some ways of doing this.

A. Configuration File Issues

Configuration files raise a number of challenges for implementing attestable systems. They can have a great impact on the behaviour of a platform, and bad settings can make otherwise trustworthy applications vulnerable to exploit. However, attestation of configuration settings is complicated, as simply providing a file hash is insufficient. Two files can have the same configuration semantics but produce different hashes, due to comments or whitespace. Our results show that a significant amount of configuration must be dealt with, and we believe that better documentation and metadata is required to describe the implications of configuration settings.

VI. Reducing Attestation Measurements

A. OS and Application Minimization

Our results come from using standard operating systems and applications. We made no special effort to reduce the number or size of programs. If we compiled a custom linux distribution, or used a smaller operating system, we would reduce the number of components and the size, moving us closer to obtaining a *trustworthy* rather than just *identifiable* system. We could also reduce the size of certain core applications. The IAIK Privacy CA project[18] uses a system trace to remove all unnecessary class files from their JVM, and a similar method could be used here. However, there is a limit to the amount of code which can be removed without reducing functionality, which may mean that minimization alone is insufficient.

B. Changing or Removing the Application Server

Glassfish libraries were the second-largest source of measurements. A smaller application server might reduce the number of libraries which must be measured. One alternative is Apache Axis2/Java, which offers similar functionality to Glassfish. However, from our experiments we found that Glassfish 2.1 used only 8 more JAR files that Axis 1.4. As an alternative, we suggest that most of the application server's functionality could be moved away from the attested platform. Much of the code required by a web service is involved in SOAP and XML parsing and handling. This functionality could be run on a separate, untrusted platform instead. Incoming SOAP messages would be handled by the untrusted platform, which would then translate and forward, using a simpler method such as Java RMI. The attesting platform now does not need to include the entire application server, thus reducing the total number of measurements.

As a one-off comparison, we restructured our attestation web service as a Java RMI server. This gave a significant reduction in measurements, requiring 78 (28%) fewer entries in the measurement log compared to Glassfish. There is a similar effect over time, saving 351 updates as well as the 78 initial measurements, 30% of the total. This difference does come at a cost, however, reducing our functionality and interoperability considerably. On the other hand, as the application server must parse lots of data in different formats, it is probably one of the main targets for a remote attack, and removing it would enhance the platform's overall trustworthiness. We intend to investigate this in our future work, and note that Wei et al[19] have taken a similar approach.

VII. Existing Work

Munetoh et al. [16] describe how to use attestation in a web services context, and suggest a number of methods for verifying remote platforms. The authors suggest that a third party Platform Validation Authority (PVA) would be used to validate attestations, using package management repositories and online vulnerability databases to make trust decisions. Their integrity database contains information for 84382 files in 1415 packages, and each platform had around 600 measurements to attest. However, so far they have concentrated on client machines, and have not considered maintaining their platform over time. Sailer at al. [10] have also implemented a similar system, as mentioned in Section II-C.

Franklin et al.[20] provide an excellent example for why servers are a good target for trusted computing technologies, demonstrating a complete, working Certificate Authority which supports integrity reporting.

Several authors have made efforts to make attestation more feasible. Jaeger et al. [21] use SELinux to control information flow, and are then able to ignore many of the untrusted applications running on a platform. This reduces the number

of measurements to verify, but does involve interpretting a complex, 1MB SELinux policy file. Paul England[8] also noted the problems with attestation, and has proposed three novel alternatives. These reduce the number of measurements to verify, at the cost of some flexibility and detail. We believe the results presented in this paper suggest that criticisms of standard attestation may have been overstated, although they remain valid for client machines rather than servers.

Haldar et al. [9] have also identified a number of problems with attestation, adding revocation to the growing list. They propose Semantic Remote Attestation, the use of a *Trusted Virtual Machine* to attest properties of code, rather than binary identities. This goes a long way to solving the *semantic gap* problem discussed in Section II-C.

VIII. Conclusion

In this paper we have shown that the package management overheads for attesting a web service are relatively small, significantly lower than most existing work has suggested. We believe that this makes attestation a viable tool for demonstrating certain platform properties, such as identifying the executables that the service intended to use, and the absence of rootkits. Overall, we think that these results give compelling evidence that *server* platforms are a good target for the integrity reporting approach.

However, the rate of change and quantity of code means that establishing platform *trustworthiness* is still not possible, especially due to the number of runtime vulnerabilities present in modern operating systems. We suspect that the ultimate solution will be achieved through operating systems and applications minimization, and increased partitioning of large, untrusted code away from important functionality.

Acknowledgment

The work described is supported by a studentship from QinetiQ. Special thanks go to Jun Ho Huh and Cornelius Namiluko for useful discussions. I am also grateful to the developers of the IMA patch for the Linux kernel and the *OpenTC* team at IAIK TU Graz for their jTSS software.

References

[1] Trusted Computing Group, "Trusted computing group home page," 2009. [Online]. Available: https://www.trustedcomputinggroup.org/home

[2] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and Implementation of a TCG-based Integrity Measurement Architecture," in *USENIX Security Symposium*, 2004, pp. 223–238. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec04/tech/sailer.html

[3] Trusted Computing Group, "TCG Infrastructure Working Group Architecture Part II - Integrity Management," November 2006. [Online]. Available: http://www.trustedcomputinggroup.org/resources/infrastructure_work_group_architecture_part_ii__integrity_management_version_10

[4] A.-R. Sadeghi and C. Stüble, "Property-based Attestation for Computing Platforms: Caring About Properties, Not Mechanisms," in *NSPW '04: Proceedings of the 2004 Workshop on New Security Paradigms*. New York, NY, USA: ACM Press, 2004, pp. 67–77. [Online]. Available: http://doi.acm.org/10.1145/1065907.1066038

[5] J. A. Poritz, "Trust[ed | in] Computing, Signed Code and the Heat Death of the Internet," in *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*. New York, NY, USA: ACM Press, 2006, pp. 1855–1859. [Online]. Available: http://doi.acm.org/10.1145/1141277.1141716

[6] U. Kühn, M. Selhorst, and C. Stüble, "Realizing property-based attestation and sealing with commonly available hard- and software," in *STC '07: Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*. New York, NY, USA: ACM, November 2007, pp. 50–57. [Online]. Available: http://doi.acm.org/10.1145/1314354.1314368

[7] D. Schellekens, B. Wyseur, and B. Preneel, "Remote Attestation on Legacy Operating Systems With Trusted Platform Modules," *Electronic Notes in Theoretical Computer Science*, vol. 197, no. 1, pp. 59–72, 2008. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2007.10.014

[8] P. England, "Practical Techniques for Operating System Attestation," in *Proceedings of the Trust 2008 Conference*, ser. Lecture Notes in Computer Science, vol. 4968/2008. Villach, Austria: Springer Berlin/Heidelberg, March 2008, pp. 1–13. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68979-9_1

[9] V. Haldar, D. Chandra, and M. Franz, "Semantic Remote Attestation - Virtual Machine Directed Approach to Trusted Computing," in *Virtual Machine Research and Technology Symposium*. USENIX, 2004, pp. 29–41. [Online]. Available: http://www.usenix.org/publications/library/proceedings/vm04/tech/haldar.html

[10] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn, "Attestation-based policy enforcement for remote access," in *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2004, pp. 308–317. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030125

[11] M. P. Papazoglou and J.-j. Dubray, "A Survey of Web Service Technologies," Informatica e Telecomunicazioni, University of Trento, Tech. Rep. DIT-04-058, June 2004. [Online]. Available: http://eprints.biblio.unitn.it/archive/00000586/

[12] The W3C, "Simple Object Access Protocol (SOAP)," April 2007. [Online]. Available: http://www.w3.org/TR/soap/

[13] K. Dietrich, M. Pirker, T. Vejda, R. Toegl, T. Winkler, and P. Lipp, "A Practical Approach for Establishing Trust Relationships between Remote Platforms Using Trusted Computing," in *Trustworthy Global Computing*, ser. Lecture Notes in Computer Science, G. Barthe and C. Fournet, Eds., vol. 4912. Sophia-Antipolis, France: Springer, November 2007, pp. 156–168. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78663-4_12

[14] E. Pelegri-Llopart, Y. Yoshida, and A. Moussine-Pouchkine, "Delivering a Java EE Application Server," https://glassfish.dev.java.net/faq/v2/GlassFishOverview.pdf, June 2007. [Online]. Available: https://glassfish.dev.java.net/faq/v2/GlassFishOverview.pdf

[15] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri, "Minimal TCB Code Execution," in *IEEE Symposium on Security and Privacy*, may 2007, pp. 267–272. [Online]. Available: http://dx.doi.org/10.1109/SP.2007.27

[16] S. Munetoh, M. Nakamura, S. Yoshihama, and M. Kudo, "Integrity Management Infrastructure for Trusted Computing," *IEICE Transactions on Information and Systems*, vol. E91-D, no. 5, pp. 1242–1251, 2008. [Online]. Available: http://dx.doi.org/10.1093/ietisy/e91-d.5.1242

[17] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell, "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments," in *Proceedings of the 21st National Information Systems Security Conference*, October 1998. [Online]. Available: http://csrc.nist.gov/nissc/1998/proceedings/paperF1.pdf

[18] A. Niederl and M. Pirker, "Build Guide for Bootstrapping a Reduced Trusted Java Compartment," March 2009. [Online]. Available: http://trustedjava.sourceforge.net/index.php?item=pca/compartment

[19] J. Wei, L. Singaravelu, and C. Pu, "A Secure Information Flow Architecture for Web Service Platforms," *IEEE Transactions on Services Computing*, vol. 1, no. 2, pp. 75–87, April-June 2008. [Online]. Available: http://dx.doi.org/10.1109/TSC.2008.10

[20] M. Franklin, K. Mitcham, S. Smith, J. Stabiner, and O. Wild, "CA-in-a-Box," in *Public Key Infrastructure, EuroPKI 2005*, ser. Lecture Notes In Computer Science, vol. 3545/2005. Springer Berlin / Heidelberg, 2005, pp. 180–190. [Online]. Available: http://dx.doi.org/10.1007/11533733_12

[21] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: policy-reduced integrity measurement architecture," in *SACMAT '06: Proceedings of the 11th ACM Symposium on Access Control Models and Technologies*, 2006, pp. 19–28. [Online]. Available: http://doi.acm.org/10.1145/1133058.1133063