Trustworthy Services Through Attestation

John Lyle

Keble College

University of Oxford

A dissertation submitted for the degree of

Doctor of Philosophy

Michaelmas Term 2010

Abstract

Remote attestation is a promising mechanism for assurance of distributed systems. It allows users to identify the software running on a remote system before trusting it with an important task. This functionality is arriving at exactly the right time as security-critical systems, such as healthcare and financial services, are increasingly being hosted online. However, attestation has limitations and has been criticized for being impractical. Too much effort is required for too little reward: a large, rapidly-changing list of software must be maintained by users, who then have insufficient information to make a trust decision. As a result attestation is rarely used today.

This thesis evaluates attestation in a service-oriented context to determine whether it can be made practical for assurance of *servers* rather than client machines. There are reasons to expect that it can: servers run fewer programs and the overhead of integrity reporting is more appropriate on a server which may be protecting important assets. However, a literature review and new experiments show that problems remain, many stemming from the large trusted computing base as well as the lack of information linking software identity to expected behaviour.

Three novel solutions are proposed. Web service middleware is restructured to minimize the software running at the endpoint, thus lowering the effort for the relying party. A key advantage of the proposed two-tier structure is that strong integrity guarantees can be made without loss of conformance with service standards. Secondly, a program modelling approach is investigated to further automate the attestation and verification process and add more information about system behaviour. Several sets of programs are modelled, including the bootloader, a web service and a menu-based shell. Finally, service *behaviour* is attested through source code properties established during compilation. This provides a trustworthy and verifiable connection between the identity of the software on a service platform and its expected runtime behaviour. This approach is applicable to any programming language and verification method, and has the advantage of not requiring a runtime monitor. These contributions are evaluated using an example e-voting service to show the level of assurance attestation can provide.

Overall, this thesis demonstrates that attestation can be made significantly more practical through the described new techniques. Although some problem remain, with further improvements to operating systems and better software engineering methods, attestation may become a trustworthy and reliable assurance mechanism for web services.

Acknowledgements

Firstly I thank my supervisor, Andrew Martin, who has provided endless insight, support and encouragement throughout my DPhil. I have enjoyed collaborating with him on several projects and papers and I am grateful for the opportunities and ideas he has provided.

I thank my parents for giving me the opportunity to pursue a DPhil and for their understanding, advice, and first-rate proof-reading abilities. I would not be here without their help and guidance over the last 26 years. I also thank my sisters for their moral support.

Throughout my DPhil I have been fortunate to study with brilliant people. In particular, I thank Jun Ho Huh, Cornelius Namiluko, Andy Cooper, Shamal Faily, Ronald Kainda, Joe Loughry, Ivan Flechais and the rest of the Security Reading Group. Our entertaining weekly discussions have contributed to this dissertation as well as my education in computer security. I am also extremely grateful to the Oxford University Computing Laboratory and Software Engineering Programme for providing several excellent courses and opportunities throughout my studies. I thank my examiners for their useful feedback and suggestions.

My friends in Oxford deserve enormous credit for making my studies such good fun. Special thanks go to George and Kate, my office mates (and fellow tea drinkers) Daniel, Tom, Peter and Meng, as well as many others in the Computing Laboratory. I thank my friends from Keble, including Richard, Ross, Tom, James, Ed, Josh and Chris, as well as and everyone at the university windsurf club. I am grateful to Steve and Alex for proof-reading this dissertation, as well as all their entertaining emails. I also thank my other friends from Weir Wood SC, Imperial College and St Paul's, all of whom deserve a mention but will, unfortunately, not get one.

My studies in Oxford have made me very grateful for the education I received at Imperial College. I sincerely thank my former tutors and lecturers in the Department of Computing.

This dissertation was funded by the EPSRC and QinetiQ. I owe them a debt of gratitude for making this project possible.

Contents

List of Figures

Chapter 1

Introduction

Service-oriented computing is a popular paradigm for implementing and designing distributed systems. Companies, governments and universities have developed grid, cloud and web services to provide access to data and for performing resource-intensive computation. There are many advantages over previous ad-hoc systems. Services can scale to match demand, charge on a per-use basis, and provide backup and redundancy. Furthermore, web service interfaces are described using open, interoperable standards, allowing them to be composed together so that complex systems can be built from many individual services [164]. This can even be automated, allowing for rapid system development.

However, the move to remote services presents new security challenges [94, 127]. Many potential users, such as pharmaceutical companies, financial services, and government departments have stringent security requirements [159, 135]. One example is scientific provenance. When processing gigabytes of data for climate models or drug trials, a key requirement is that researchers should be able to trust the result of remote computation [86]. If the computer that ran the experiment was insecure, it could be tampered with to produce incorrect results. This could reduce accuracy and cost time, money and the researcher's reputation. Unfortunately, the motivation for attacking and compromising these systems exist, as the recent 'Climategate' scandal has shown [30], and mechanisms are required for protecting these systems. Users need the ability to establish the trustworthiness of remote services despite the presence of motivated attackers.

For the purposes of this dissertation trustworthiness is defined in terms of behaviour. When users seek assurance of a service, they aim to make sure that it will *behave* in the manner they expect. This means that security requirements are met, and that more general integrity guarantees hold, including the behaviour of an algorithm, or the reliability of storage. The aim is to go from services which are *trusted* – relied upon without any supporting evidence – to *assured* – relied upon because of unforgeable evidence of their behaviour.

A crucial part of the problem is that these systems are *remote*, and will be running software that users cannot assess themselves. Most existing approaches for gaining trust in remote systems therefore rely on the ability for each platform to *attest* to the software and hardware

it is using. This seems essential, as without knowing what the platform consists of, how can its behaviour be known, let alone trusted? However, the *security* of the remote attestation mechanism now becomes an important challenge, otherwise an untrustworthy computer can report that it is running trustworthy software.

Fortunately, trustworthy attestation is part of *trusted computing*, a technology which has been available for several years. It uses tamper-resistant hardware to report on the state of software. If users trust the manufacturers of the hardware, they can then believe what it says about the software, and therefore the state of the machine itself. Theoretically, the combination of software assurance methods – such as testing and verification – along with attestation should make assessment of remote platforms possible. However, few companies and services use attestation today. There appear to be significant practical problems, primarily in the semantic gap between establishing what software has been *run* on the platform and whether or not it should be *trusted*.

This dissertation explores the problem of attestation for establishing trust, and answers the following questions:

- To what extent is remote attestation a practical solution for web service assurance?
- What are the key problems, and how significant are they?
- Can it be made more feasible through new tools and software engineering techniques?

In the following section, the motivation for answering these questions is discussed, and several example situations are described. In Section 1.2 an outline of the dissertation is provided, as well as a summary of the the main contributions.

1.1 Why Do We Need Trustworthy Services?

Before diving into the main thesis question – to what extent attestation is a feasible mechanism for gaining assurance in services – it is worth considering whether there is any real *need* for trustworthy, high-assurance services. Are there situations where an unreliable or insecure service would have a significant impact? The focus of this dissertation is on assurance in terms of *security* behaviour but algorithmic behaviour, or even formal *correctness* is just as important. Terminology is discussed in more detail in Section 1.3. The following four sections give examples of scenarios where service-oriented computing is in use, and trustworthiness is critical.

1.1.1 Services for healthcare

Medical science and the heathcare industry have been enthusiastic adopters of service-oriented architectures. Research projects in medicine have significant security requirements to maintain the confidentiality of patient data [168]. Furthermore, hospitals and clinics would benefit from greater information sharing through standardised XML interfaces [90], but have obvious

concerns about the integrity and confidentiality of this data. Being able to demonstrate the trustworthiness of these systems is increasingly important in light of laws such as the US Health Insurance Portability and Accountability Act [226] (HIPAA) and The Data Protection Act and Caldicott report [52] in the UK. Indeed, these last two specify that 'Suitable security should be in place to protect data' and that 'Information access should be on a strict need to know basis.'

These security requirements are a challenge to implement, and put a burden on healthcare administrators to make sure that the computer systems being used are protecting data properly. This implies the need for auditing and evaluation of all services that are relied upon. Mechanisms such as attestation may help provide evidence that private information is not being leaked to unauthorised parties.

1.1.2 Electronic voting

Electronic voting systems have clear security requirements. The goal of these systems is to allow online voting which can be potentially cheaper, easier and more reliable than requiring voters to turn up at polling stations. There are advantages for voters with disabilities, for whom getting to a polling station is prohibitively difficult and for people who usually have to fill in a postal vote. Indeed, Estonia used electronic voting for local government elections in 2005 [225] in order to try and increase turn-out.

However, voters must be able to trust the voting machines to properly record their vote, keep it confidential and produce the correct tally at the end of the election. This turns out to be a difficult problem and the subject of a great deal of academic literature [44, 157]. Attestation of electronic voting systems has been considered and Gardner et al. [68] convincingly argue that software-based attestation systems are likely to be inadequate for electronic voting systems. Alternatives using trusted computing may solve some of these problems but must be made practical for voters.

1.1.3 Financial services

Web services are often used in financial institutions [235] to process incoming information such as trades, transactions and stock quotes. The inherent interoperability is attractive for companies that work together with this information. However, the incentive for attacking these services is clear and it is reasonable to expect users to be wary of remote systems which process information such as credit card details. Financial institutions have suffered from software-based attacks. The Heartland Payment System [109] was used by over 250 thousand businesses to process 100 million credit card transactions every month. Malicious software was installed on the payments system, which then stole potentially tens of millions of card details. Similar incidents at RBS Worldpay and Hannaford Bros., also involved malicious software [109]. Secure attestation of software identity information might have helped avoid these attacks.

1.1.4 Provenance of grid services

Computational and data grids are used by scientific institutions to distribute complex application workloads and to share experimental results. However, the quality of these results depends on the trustworthiness of all of the computers in the grid. Some scientific data are known to have security concerns, such as private healthcare records and politically-sensitive climate models, and almost all experimental data require high integrity storage and processing. As a result, data *provenance* in the face of potential adversaries has become increasingly important [86] particularly in service-oriented architectures such as grids [203].

> 'From an operational perspective, there is a need to provide solutions that are secure, reliable, and scalable. Scientists need to be able to trust that their input and output data are secure and free from inappropriate data access or malicious manipulation.' [73]

One of the key threats is that provenance information could be *forged* to make fake data appear reputable. Therefore, a requirement for secure provenance is that participants can provide tamper-proof evidence of how data were acquired and processed [86]. Attestation could provide some of this evidence, assuming it can be made practical and easy to validate. More details on this approach can be found in Appendix C.

1.1.5 Summary

The four examples – healthcare, voting, finance and scientific provenance – demonstrate the need for trustworthy and *attestable* services. Users must be able to check that the service they are trusting with their account details, vote or healthcare data will behave as they expect, and not betray any of their sensitive information. Trustworthy attestation is a good starting point for providing this functionality, as it can provide the identity of the software being used by these remote platforms. If this can be combined with software assurance methods, to show that the attested software will work in the correct manner, then a financial service would be able to demonstrate in a believable way that it will implement the functionality promised in its service description. Realising this goal has the potential to make service-oriented computing more practical for both users and providers [164].

1.2 Contributions and Dissertation Structure

The main contributions of this dissertation are an analysis of the feasibility of using attestation for web service assurance, and a range of techniques that have been developed for making it more practical and trustworthy. The next section looks at terminology, to clarify ambiguous language and define concepts for the rest of the dissertation. Chapter 2 provides background material on trusted computing and web service concepts, as well as summarising methods for software assurance. The strengths and properties of these are compared, and the necessity of combining them with attestation in a service-oriented scenario is established. Chapter 3

presents a list of the open problems with attestation, and the approaches taken in existing research to overcome them. Remaining issues with attestation that have not been adequately solved are discussed in a gap analysis, which provides motivation for the content of the following chapters.

The first novel contribution is made in Chapter 4, where the problem of service attestation is analysed and quantified over a two-and-a-half year period by applying software updates to an example platform. This is accompanied by a review and comparison of attestation approaches and the most practical situations for its use. One of the problems identified – the large trusted computing base of a service – is immediately tackled in Chapter 5. The proposed solution is to move the core service implementation to another platform, and the various problems of doing so are then discussed and mitigated.

Having addressed one issue with attestation, Chapter 6 moves on to the problem of interpreting platform state. Existing specifications do not make it easy to use information gained from attestation to establish what state the platform is in. A layer of abstraction is described which allows programs to be modelled and composed together to form a system which explains the attested information. This also provides a framework for developing applications with attestation of behaviour in mind.

In Chapter 7 the next step is taken, to allow developers of services to attest to not only the identity of their software, but also code-level properties such as invariants and post-conditions. This, in combination with the application framework and minimal service infrastructure, allow for the creation of an *attestable* electronic voting ballot box service, which is analysed as part of the evaluation in Chapter 8. The evaluation considers to what degree the problems with attestation identified in Chapter 3 are solved by the new system. Finally, in Chapter 9 future work is proposed, and the dissertation concludes.

Supporting material is provided in the appendices. Appendix A gives further details about the algorithms used in the electronic voting service. Section B.1 of Appendix B gives an example build script for compiling part of this service and demonstrates how integrity measurement can be integrated into the build process, as discussed in Chapter 7. Section B.2 then gives the script used to run Prolog models from Chapter 6 and the evaluation. Appendix C is an extract from a related paper on applying attestation to provenance, which provides motivation for the arguments and results presented in this dissertation.

1.3 Terminology and Definitions

There are several ambiguous concepts in computer security and assurance, and this section covers how they will be used for the rest of this dissertation. Some additional definitions are given in the glossary. Where possible, the notation and terminology defined is justified by existing work.

1.3.1 Trust and trusting software

The words 'trust' and 'trustworthiness' are difficult to use precisely. In general, the notion of trust refers to a belief in the behaviour of something. Garfinkel et al. [69] use trust to describe 'our level of confidence that a computer system will behave as expected.' This is similar to the Trusted Computing Group's definition that a trusted system is one where 'hardware and software behaves as expected' [210]. Trustworthiness, on the other hand, is about whether this trust is well placed. A trustworthy person is *able and willing* to act in the best interests of the trusting party [133]. This can be established by having some evidence (*assurance*) that this statement is true. For example, having experience of good behaviour in the past might make someone reasonably trustworthy for the future. RFC 4949 [192] defines a trustworthy system as one 'that not only is trusted, but also warrants that trust because the system's behaviour can be validated in some convincing way, such as through formal analysis or code review.' This formal analysis or code review is the *assurance* method.

Unfortunately, no single reference provides a consistently useful set of definitions for trust and trustworthiness. For the purpose of this dissertation, therefore, when the following phrases are used, they will have the given meanings:

Alice can be trusted. 'An entity can be trusted if it always behaves in the expected manner for the intended purpose' [160]. This means that trust is about behaviour, and not just *security* behaviour, such as the confidentiality of data.

Alice trusts Bob. Alice believes that Bob will behave as expected. There could be any (or no) reason for this belief. Generally the notion of trusting an entity only occurs when the entity has the ability (but hopefully not the intention) to abuse that trust. For example, if it is given a private key or has a security function. In this dissertation, *trust* can be thought of as synonymous with *faith*.

Alice must trust Bob. Alice needs Bob to behave in the expected manner. Alice may or may not have any reason to believe that Bob will.

Trustee and Trustor. In a one-way relationship where Alice trusts Bob, Alice is the trust*or* and Bob is the trust*ee*. In the majority of this dissertation, the trustee will be a service computer platform, and the trustor will be an end user / service requester.

Alice is trustworthy. Alice *will* behave as expected, and the trustor is correct to trust her for the intended purpose.

Establishing trust. While this seems to contradict the other definitions, 'establishing trust' is the process of working out whether an entity is *trustworthy*, that is, whether there is any *reason* to trust it. This phrase could more accurately be 'establishing trustworthiness,' but much of the literature uses the former phrasing.

Entity X is trust*able*. Sufficient infrastructure and information exists that a decision about whether entity X is trustworthy *can* reasonably be made. This would not be the case, for

example, when a remote party is unable to identify who or what entity X is, or has no idea what it is supposed to do. An important distinction is that Entity X may be trustable, but not trustworthy. The first two of Proudler's three steps for establishing trust (see Section 1.3.3) are used in order to make the entity trustable, then the third is used to make the decision on trustworthiness.

Assurance The process of building evidence to show that something is trustworthy. Attestation is a form of assurance, as are code reviews, certification and testing.

Security and Trust As Gollmann explains [75], misusing the term 'trust' can have bad consequences for security. Security has a different meaning to trust, and is defined as a *subset* of trustworthiness [77]. A trustworthy entity is by definition secure, as it will behave 'as expected' with respect to the user's security requirements. However, a security requirement may significantly raise the bar for the trustworthiness of a system, as there may be known, motivated attackers. Because of this, the security of the system may be one of the most difficult and important aspects, and the security of the *assurance* process may also be vital. This is generally the case for systems proposing to use trusted computing technology.

These definitions can apply to many things, including people and objects. However, is the notion of trust relevant to software and electronic systems? It is more usual to discuss these with reference to a specification, using formal proof to establish correctness rather than trustworthiness. However, there are many reasons why the less precise definition of trust is more applicable. In part this is due to the problems associated with specifying and verifying software [190]. It is considered infeasible to specify large programs, such as operating systems, making it impossible to verify their correct implementation. Moreover, almost all applications rely upon several of these large systems, so while they might be correctly implemented, the assumptions they make may not hold. Worse still, applications themselves are almost never formally verified, and generally contain bugs. This means that software cannot be considered to match the precise intentions of the programmer. These bugs may be security vulnerabilities, which means that the software an end user interacts with may have been altered by a malicious party. The original programmers may have also intentionally introduced undesirable behaviour into an application, which the user may not know about. Almost all software is therefore capable of silently betraying its user, despite previous experience. As a result, the behaviour of software is effectively unpredictable. Indeed, from the users' perspective, the majority of applications are sufficiently complex that most people perceive their computers as *social actors* [238] rather than deterministic, predictable algorithms. This makes sense from a programming perspective as well, because many systems are so large that no single person understands all of the code. Overall, therefore, trustworthiness can be used as a reasonable pragmatic alternative to correctness, and one that makes sense when thinking about the perspectives of those using and developing computer systems.

1.3.2 Trusted computing base (TCB)

The Trusted computing base (TCB) has been defined by the US Department of Defence [227] as

> 'The totality of protection mechanisms within a computer system – including hardware, firmware, and software – the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system. The ability of a trusted computing base to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (e.g., a user's clearance) related to the security policy.'

The same definition is used in RFC 4949 [192]. An important feature of the TCB is that it is the minimum amount of code and hardware that *must* be trusted, and any flaw within it has the potential to invalidate the security of the entire system. However, this quotation only describes a TCB in terms of system security. A correct, security policy-enforcing TCB will not be able to guarantee overall behaviour, which contradicts the earlier definition of trust. As such, a TCB from this definition is necessary but not sufficient for behavioural trustworthiness, assuming the expected behaviour of the system includes the non-violation of the enforced policies.

For the purpose of this dissertation, a TCB is defined as:

> 'any part of the system, including hardware, firmware and software, which must be operating correctly in order for any defined behavioural properties of the system to be met.'

This description and the earlier one might be reconciled if 'security' includes a very broad notion of integrity. An example illustrates this point. On a consumer banking website, a classic TCB would include all the software responsible for authenticating users, preventing one customer from viewing and modifying the accounts of another, and so on. The definition used here, on the other hand, includes those points but also that any standing orders a customer creates will work as expected, and that suitable warnings are given before transactions are carried out. This is a subtle point, but important when using a Trusted Computing Group (TCG) definition of trust.

1.3.3 Proudler's three steps to establishing trust

Proudler gives three requirements for establishing trust in an object or entity [169]:

1. It must be correctly and unambiguously identified.
2. It must be operating unhindered.
3. The trustor must have experience (or trust someone who has) of its consistent previous good behaviour.

The first step means that the entity being trusted can be reliably identified by the trust*or*. Without this, a similar entity with different behaviour might be able to impersonate it. For people, the solution to this problem is (on a small scale) simply recognising someone's physical appearance. For software, this might be a hash of the executable, or the assumed confidentiality and ownership of a private RSA key.

The second step means that the entity is behaving in its usual manner, with no outside influence or extreme condition altering its behaviour. For a person, this might not be the case if they were being blackmailed, or intoxicated. For software, infection with a virus, or corruption of part of the binary are examples of hindrances.

These two steps make an entity trust*able*. That is, they allow a trust decision to be made, as the entity can be identified and assumed to be working as normal. The final step is to establish what can be expected as *normal*. In the case of a simple software calculator, this might mean that numbers have successfully been multiplied together in the past, leading the trustor to believe that it will happen again in the future. Importantly, *good* behaviour does not necessarily mean *correct* behaviour, as a calculator that is known to make small rounding errors might still be trusted at a lower level of precision. Similarly, a more complicated piece of software with *known* bugs may be trusted to always reproduce these bugs. More discussion of the merits of previous experience are discussed in Section 2.1.1.

1.3.4 Security and assurance properties

The term *security property* is frequently used in the literature, but poorly defined. Most commonly, a security property is one of the standard information security components: confidentiality, integrity, availability and, sometimes, accountability [195]. These can be refined further, into protocol properties such as non-repudiation, agreement, non-interference and authentication [63]. Separability is another example, as the property that 'no interaction is allowed between high level and low level events' [241]. However more specific code-level properties are sometimes discussed. The MOPS tool [34] checks for temporal properties including 'any call to chroot should be immediately followed by a call to chdir("/")' and 'a privileged process should drop privilege ... before calling execl.' Sadeghi and Stüble [178] discuss property-based attestation, and by properties they 'informally mean a quantity that describes an aspect of the behavior of that platform with respect to certain requirements, e.g., a security-related requirement.' The example given is of a 'secure operating system providing isolation of processes or confinement etc.' Poritz et al. [166] give examples of attestable properties as 'the absence of certain vulnerabilities or the ability to enforce certain policies' as well as 'privacy and availability statements.' They go on to include conformance with common criteria, and uptime requirements.

This dissertation is concerned with *assurance* and *trustworthiness*, which refer to behaviour beyond just security properties. For this reason, *assurance properties* will be the focus. Assurance properties are aspects of the system that can be established through evidence provided by the system, or a third party, through techniques such as attestation. These aspects may include security properties, behavioural properties, or statements about the platforms involved. They

will make guarantees of different strengths, and may rely on other assumptions. For example, the presence of an anti-virus program might provide assurance. If every computer on a network reports on whether anti-virus is active then this might provide a guarantee that files with known virus signatures will be prevented from running. However, it does rely on the trustworthiness of the reporting mechanism, and does not provide any information about protections from other attacks, or any detail about the specific behaviour of the platform. Assurance properties are therefore the *reportable* facts about a system that can inform a relying party about its trustworthiness.

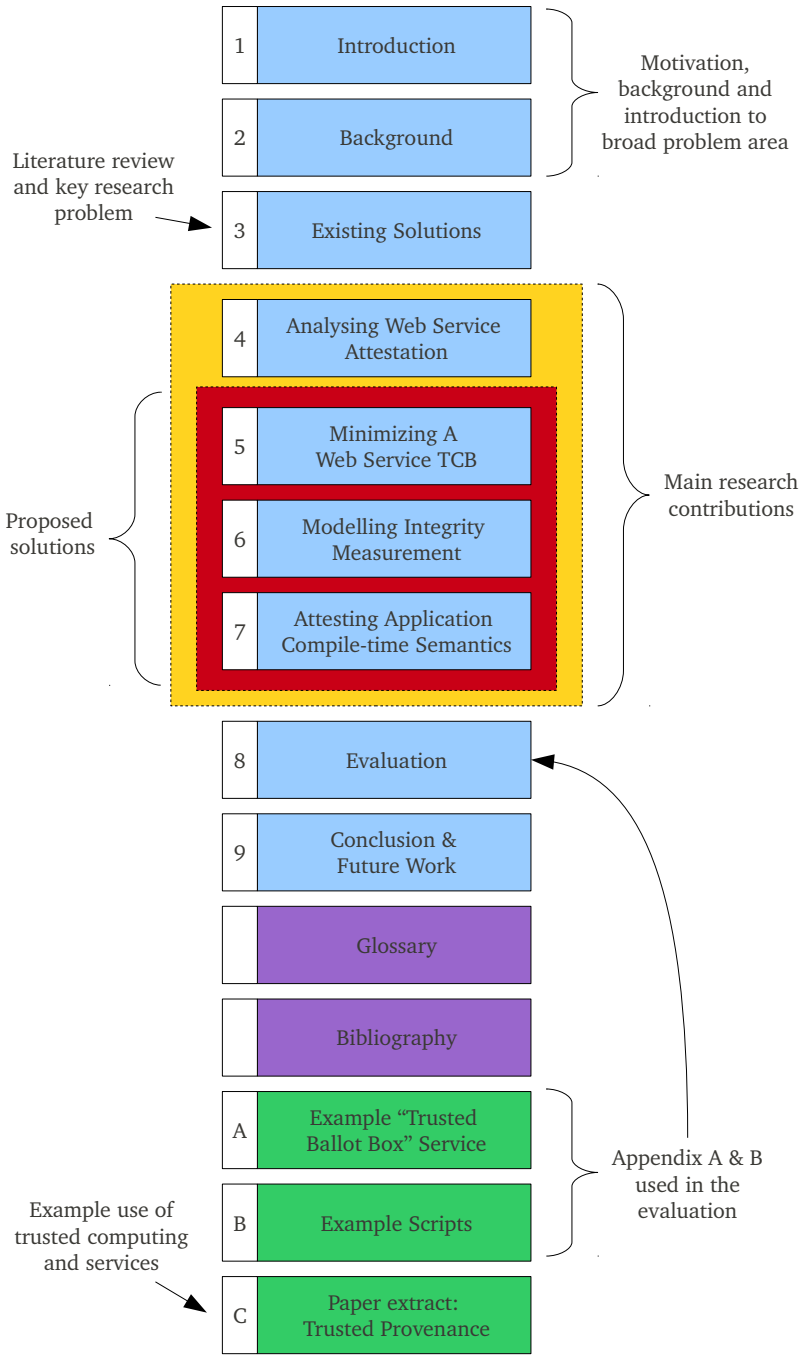

Figure 1.1: Dissertation structure

Chapter 2

Establishing Trust in Software Systems

The need to assess and evaluate the trustworthiness (and security) of computing platforms has been around for nearly as long as computing platforms themselves [233, 26]. The examples in Section 1.1 describe some of the reasons for assessing online services in particular, but a great deal of research has gone into evaluating all kinds of software and hardware. This chapter begins in Section 2.1 with a brief, high-level overview of some approaches to software assurance.

Following this, Section 2.2 introduces service-oriented architectures, and the terminology used to discuss them. This discussion highlights an important difference between the assurance of services, and most of the systems discussed in the literature on software assurance. A web service user may not have access to the source code, the installed applications, operating system, or hardware. Users will not necessarily know the identity (or even the developer) of the software at the endpoint. This is a problem, as Proudler's three requirements for trust (see Section 1.3.3) state that identity is one of the pre-requisites. Attestation, as part of trusted computing, is one potential solution and is the focus of the research presented in this dissertation. Section 2.3 provides a comprehensive overview of trusted computing features, and the kind of assurance it can provide for remote platforms.

Assuming attestation can provide basic assurances, it is necessary to look at how more detailed information about software behaviour might be established. This includes techniques such as static analysis and proof-carrying code, as well as specific methods used for web services. This is covered in Section 2.4, and then the chapter concludes in Section 2.5.

2.1 System Assurance: An Overview

The following approaches are commonly used to establish the trustworthiness of a software system, or avoid the risk associated with using it.

2.1.1 Previous experience and reputation

Perhaps the most logical method of gaining assurance in software is to use it, and observe whether it works as expected. If it does, then it may be considered trustworthy in the future. This is step three of Proudler's method for establishing trust: having experience of consistent good behaviour in the past.

However, there are problems with relying on experience. It will never exist the first time a piece of software is used, and there are situations where it is impossible to build it up. For example, when trusting the air bags in a car, or a new online banking system. Secondly, online software is liable to change, and good behaviour in a previous version does not imply good behaviour in the current one. In fact, the same piece of software may work on one day and not the next, perhaps because of an increased load, user input or another interfering program. Previous experience is equivalent to black-box testing, and has the same fundamental, well-known problem [59] when attempting to use it to establish the *absence* of bugs. Therefore, it may be impossible to ever build up valid experience of a remote software system.

These problems might be avoided by using the previous experience of another person. However, this requires the trusting party to trust this other person, and eventually someone is going to have to try the software for the first time. Reputation-based systems try to take some of these factors into account, but all have their own limitations or overheads. As a result, it is likely that many users will be unwilling to use someone else's experience to chose a critical system.

2.1.2 Certification

Military and government projects often use certified software in an attempt to increase the level of assurance. In security, for example, the Common Criteria ISO standard has seven evaluation assurance levels, with higher levels implying more thorough evaluation. The certification does not, however, necessarily imply better security, and there are several problems [125] with relying on it. There is a wide range of other evaluation and certification standards, discussion of which is beyond the scope of this dissertation.

2.1.3 Source-code analysis and verification

A more traditional way of establishing whether or not a piece of software will behave properly is through formal verification. This requires a formal specification of the behaviour of the system, followed by an analysis of the code to see if it conforms. There is a huge range of literature surrounding this topic. A complete survey has not been provided due to the size of the field, but several relevant techniques are discussed in Section 2.4.

2.1.4 Using a high assurance software development process

The software development process used can also indicate the expected quality of an application [6]. One reason for looking at this is that it is extremely difficult to spot bugs in the

software itself, but the process could show how many are likely to exist. Good software engineering practices are believed to improve the overall quality of software, although there is some dispute over what these practices are. Some metrics can be used to quantify the quality of software, both later on and early in development [104]. These include factors such as code complexity, design purity [13], test coverage, bug reporting statistics and the rate of change of the code base. Other practices are harder to measure, such as the requirements capture method and team communication. These metrics are also quite difficult to analyse sensibly, as different development practices may suit different projects. The key problem is summarised by Amoroso et al. [6]:

> 'characterizing the ideal software process is especially impractical, in light of the fact that the software engineering community has yet to develop practical, widely accepted techniques for developing software that is free of errors'

Furthermore, it is important to emphasize that even if reliable software quality metrics were established this would still not be useful for a *remote* relying party. A remote party cannot reliably establish the *identity* of the running code and would not be able to believe any reports about its development method or the result of any verification. This is why trustworthy attestation is required: to find out exactly what software is running on a remote system such as a web service.

2.2 Service-Oriented Architectures

Service oriented architectures (SOA) provide the context for this dissertation, as they are a popular way of creating distributed systems with an emphasis on standards and interoperability. They have been adopted by a wide variety of companies, as well as governments and academic institutions, as they provide common interfaces for different data sources and functionality. A definition of the overall concept of service-oriented computing is given by Papazoglou and Dubray [156] :

> 'Service-Oriented Computing (SOC) utilizes services as the constructs to support the development of rapid, low-cost and easy composition of distributed applications. Services are self-contained processes – deployed over standard middleware platforms, e.g., J2EE – that can be described, published, located, and invoked over a network'

Individual web services themselves are arguably just networked applications which can communicate in a standard format and perform some valuable function. Some services are used to make existing (or proprietary) databases available to other systems. They can also be more complex; one service might contact several others and be part of a much larger overall transaction. Web services have been defined by many people in different ways, but for the purpose of this report the following one is used [156]:

> 'A web service is a platform-independent, loosely coupled, self-contained programmable web-enabled application that can be described, published, discovered, coordinated and configured using XML artefacts for the purpose of developing distributed interoperable applications. Web services possess the ability to engage other services in a common computation...'

2.2.1 Components

There are several components in a standard service-oriented architecture, with the following definitions taken mostly from Singh and Huhns [195]:

Service Provider. The provider creates and operates the web service. They also advertise it to potential users by registering it with service brokers.

Service Brokers. They maintain a list of all services that have been registered with them. They may also provide other functionality for service discovery. Typically the service broker holds the interface definition of each service.

Service Requester. The end user of a service. This might be a person, piece of software or another service. They search the registry to find a suitable provider.

2.2.2 Standard technologies

Web services rely on various standards, most of which use XML. SOAP [215] is used for exchanging messages between the service provider and requester. It defines the structure of the content, as well as encoding rules. SOAP messages are routed between recipients until they arrive at the final destination. This may include any number of intermediary nodes. SOAP also defines fault elements which can be sent by the service in a number of situations, including incorrect client request formats or internal errors.

The Web Service Description Language (WSDL) is another standard based on XML. It is used to describe the programmatic interface of a web service, in terms of its address and method signatures. This includes information on data types, arguments and return values.

The Universal Description, Discovery and Integration (UDDI) specification is used to register and locate web services. It defines a registry where organisations can describe themselves and publish their services so that potential clients can discover them. UDDI is itself a web service based on XML and SOAP [195].

2.2.3 Dynamic selection

While each *individual* web service may offer some kind of useful functionality, the real benefit of SOA is that they can be combined together easily, allowing the rapid creation of new, custom applications. These workflows also have the potential to be very reliable, as services can be chosen and composed together at the last minute. This means that an individual fault can, when detected, be dynamically avoided by choosing alternative services where necessary [196].

However, many of the perceived advantages rely upon better specification and assurance of the component services. Without knowing precisely how each will behave, it is difficult to use them in combination with any confidence [164]. Testing web services is also difficult, as they might exist in different administrative domains or operate on live data. This becomes more of an issue when considering services with critical functionality, such as in financial, medical or valuable intellectual property scenarios. Remote verification of web services therefore seems necessary, but few methods of doing so have been developed.

2.2.4 Common threats and vulnerabilities

To gain assurance in a service-oriented architecture, there are a number of threats to consider. From the requester's perspective, the service provider has the best opportunity to betray secrets or make a service act maliciously. Even assuming the provider is largely trustworthy, there have also been numerous examples of disgruntled employees abusing their privileged status to attack their systems [76]. This kind of *insider* attack might be carried out through malicious software, a modified script, or an unencrypted communication channel. One of the advantages of trusted computing is that malicious software can be identified through attestation before the service is used. Furthermore, TPM protected storage can prevent data leaks even with privileged access to the system.

However, service-oriented architectures also face many threats from outside. A malicious party might attack the system to steal customer data, or to alter its behaviour. The following threats are defined by Bhalla and Kazerooni [18] and The National Institute of Standards and Technology [197] as being particularly important. Assurance of services must therefore focus on guaranteeing that the running software will be robust despite these threats and vulnerabilities.

- Message alteration, falsification and replay.
- Loss of confidentiality. Information within a message being disclosed to an unauthorized individual
- Forged credentials. An attacker makes a request to the service using stolen or fake credentials.
- Denial of service.
- Exploiting XML parsers and validators. The XML parser or validator may contain a buffer overflow, or be vulnerable to denial of service through input of a large file with complex data structures.
- Error handling. Presenting too much error information can make an attacker's job easier, highlighting a potential SQL injection, for example.
- XPath or SQL injection. Both XPath and SQL queries can be designed to return more information than was anticipated, avoid access controls, or to execute arbitrary statements.

Throughout this dissertation, these threats will be considered with respect to assurance. Any proposals that might create a vulnerability to one of these threats will be identified and mitigated where possible.

2.3 Trusted Computing and Virtualization

Remote computing platforms and their software are currently not trust*able*: it is not even *possible* to reliably establish whether or not they are trustworthy (see the definition in Section 1.3.1). Only weak identification methods exist for the endpoint – IP and MAC addresses, user credentials – and there is no way of reliably finding out what software a remote computer is running. The platform can be queried, but nothing prevents a malicious system from reporting falsities. Furthermore, viruses and trojans make otherwise trustworthy software behave in untrustworthy ways. This means that a remote system cannot be relied upon to *say* that it is trustworthy. This is because computer systems are fundamentally unrestricted – they can be programmed for *any* purpose, and can run potentially any software designed for it. While this flexibility is partly responsible for the successful history of computing, it means that software is always capable of working against the user's intentions. At every level, software could betray its user – malicious applications, operating systems and firmware all exist. Therefore, mechanisms in *hardware*, a less malleable medium, are necessary to protect against malicious software, and provide evidence to support the honesty of the platform's interactions.

Trusted computing is a paradigm developed and standardized by the Trusted Computing Group [214], based on exactly this principle. It aims to enforce trustworthy behaviour of computing platforms by identifying a complete *chain of trust*, an ordered list of components on a system that are relied upon for trustworthy behaviour, including all hardware and software. Assurance of each link in the chain is dependent on the trustworthiness of every earlier component. 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. This approach is known as *integrity reporting*.

2.3.1 The Trusted Platform Module (TPM)

The technologies proposed by the TCG are centred around the Trusted Platform Module (TPM). In a basic server implementation, the TPM is a chip connected to the CPU. It can securely store RSA keys, and holds a unique private key (the *endorsement key* or *EK*). It also contains at least 16 Platform Configuration Registers (PCRs). These are reset at boot and can then be read by software. They can only be written to in one way, through the `extend(..)` operation. This updates the PCR value to be the SHA-1 hash of the old value along with the new data given as an argument to the operation. Therefore, at any time a PCR value will be of the form

$$pcr_m = SHA1(A_n \mid SHA1(\ldots SHA1(A_1 \mid SHA1(A_0 \mid 0\text{x}00))))$$

where $A_0..A_n$ are all the values extended into PCR number m and $SHA1(x \mid y)$ computes the hash of x concatenated with y. Most PCRs start with value 0. Separately from this, a log is kept of the actual $A_0...A_n$ values, and this log can be verified against the final PCR value by recreating the entire hash chain. In this way, one PCR can record a long chain of hashes.

2.3.2 Authenticated boot

The limited functionality offered by the TPM is ideal for recording the boot process of a platform, with 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. 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.

Kauer [103] gives three properties of this chain of trust which must hold for the system measurements to be trustworthy:

1. The first code running and extending PCRs after a platform reset (the SRTM, see Section 2.3.4) is trustworthy and cannot be replaced.
2. PCRs are not resettable without passing control to trusted code.
3. The chain is contiguous. There is no code in between that is executed but not hashed.

This authenticated boot functionality is useful to the owner of a system, as they can check that no viruses or root kits were loaded at start up. However, it might also be interesting for a remote user. For this reason, the TPM also contains a mechanism for reporting the PCR values in a tamper-proof manner, called *remote attestation*.

2.3.3 Remote attestation

The TPM can create a signed copy of its PCR values. This can be given to a remote party (the 'challenger') for inspection, along with the Integrity Measurement Log (IML), recording the application hashes that have been extended. 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). The public half of the key must be certified by a third party certificate authority (a *Privacy CA*) which confirms that a real TPM holds the private half. The reason for this additional key is to preserve platform privacy – an AIK certificate shows only that the platform *has* a TPM, not which one it has. Multiple AIKs can be created for the same TPM. Full details can be found on the TCG website [214], and a nonce-challenge attestation protocol has been specified by Sailer et al. [180]. An alternative to using a Privacy CA is Direct Anonymous Attestation (DAA), which preserves the privacy of the attesting party through use of a zero-knowledge proof [24].

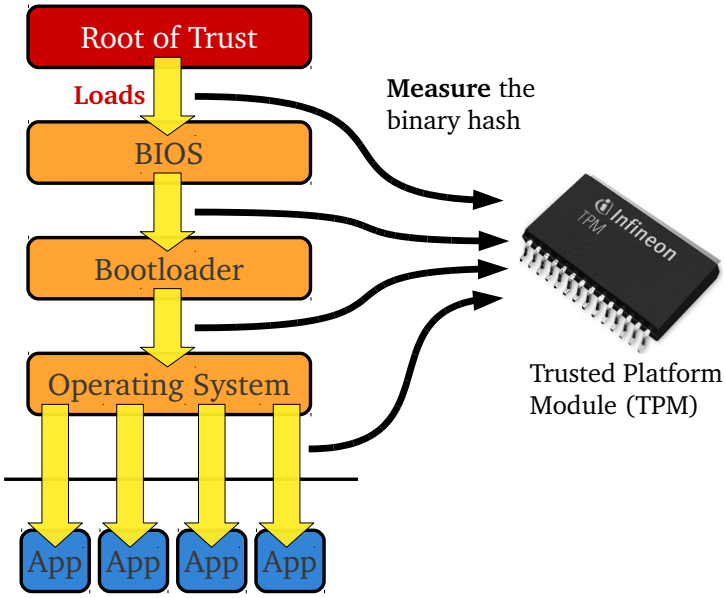

Figure 2.1: Authenticated boot

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 [207] (RMDB). 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: deciding whether or not a certain configuration is trustworthy. This is an open problem in trusted computing research.

2.3.4 Roots of trust

The TCG define a *root of trust* as:

> 'A component that must always behave in the expected manner, because its misbehavior cannot be detected. The complete set of Roots of Trust has at least the minimum set of functions to enable a description of the platform characteristics that affect the trustworthiness of the platform.' [210]

An example is the Root of Trust for Measurement (RTM) which is the first element in the authenticated boot process. The TCG define it as 'a computing engine capable of making inherently reliable integrity measurements' [210]. The RTM begins integrity measurement by measuring itself. It then measures the next element in the boot process, and passes over control. On a standard server or laptop, the static RTM (SRTM) is the first sector of the BIOS. It is a *static* measurement because it takes control immediately after platform reset, and its

self-measurement can only be triggered by this event. Furthermore, it is a small block of functionality that should never require modification.

Other roots of trust include the Root of Trust for Reporting (RTR) and Root of Trust for Storage (RTS). The RTR is an 'entity implicitly trusted to report information accurately and verifiably to outside entities' [78]. It is responsible for implementing secure attestation. The RTS can 'be trusted implicitly to store information without any interference or leakage' [78]. It implements the key storage and sealing mechanisms discussed in Section 2.3.6. Both the RTR and RTS are provided by the Trusted Platform Module.

More recently, other items have been proposed as roots of trust. St Clair et al. [198] propose the root of trust in integrity, or 'root of trust installation' (ROTI) which links all software installed on a platform to the original program, the ROTI, which installed and configured it. This is discussed further in Section 7.6. Cabuk et al. [29] have also described a 'Software-based Root of Trust for Measurement (SRTM)' which is not quite a *root*, as its integrity is established and reported through the hardware roots described previously.

2.3.5 Dynamic root of trust for measurement and late launch

The dynamic root of trust for measurement (DRTM) is an alternative to the static RTM. It can be run at any time after platform boot, allowing an untrusted platform to launch a virtual machine (or any piece of code) which will be measured, starting from the dynamic root, without any interference from the software currently running. On Intel processors, the DRTM is implemented by the `SENTER` CPU instruction, and `SKINIT` on AMD chips. This entire process is known as *late launch*.

On an Intel platform late launch occurs when a component loads the MLE and SINIT modules into memory and issues the `GETSEC [SENTER]` command [79]. The MLE (Measured Launch Environment) is a trustworthy piece of software, typically a virtual machine monitor, capable of running isolated virtual machines. SINIT is another software module, responsible for measuring and launching the MLE. After `GETSEC [SENTER]` is called, the processors are synchronised (so only one is left running) and external event handling is stopped, disconnecting DMA and interrupts. Next, all the bytes of PCRs 17-20 are reset to 0x00 and a hash of the SINIT module is extended into PCR 17. The SINIT module is executed, and tests for proper hardware configurations. It then measures and loads the MLE, re-enables external events and executes the MLE code. When ready, the MLE can then run `GETSEC [SEXIT]` to re-enable all other processors. The end result of this is that MLE code has been run without any interference, and has been measured into a PCR. This is then the base for running operating systems and applications which support integrity measurement.

Attestation of a dynamic root of trust involves a TPM Quote of at least PCRs 17-20. The PCRs have all their bytes set to 0xFF at boot time, and then are reset to 0x00 at late launch. Imitating this would involve calculating Q such that 0xFF= $SHA1$(0x00|| Q). This is considered infeasible. After checking that the PCRs were reset, the hashes of the SINIT and MLE must be checked, and then the IML is verified as per normal.

The advantage of this approach is that the BIOS and bootloader do not need to be measured.

This is particularly useful as the BIOS will load (in an effectively random order) many Option ROMs. These are from many different manufacturers, and a SRTM system relies on all of these being trustworthy. This is considered unrealistic [79]. For this reason, the OSLO boot loader has been developed which uses the DRTM rather than relying on the earlier boot process [103].

2.3.6 TPM protected storage

Another feature of the TPM is that it can `seal` or `bind` arbitrary data to PCR values. This allows data to be encrypted to one specific TPM and only allow decryption when its PCRs have a particular trustworthy value. This might be used to prevent a certain document being opened by anything other than a trusted reader application. One way this is implemented is by creating a TPM sealed key. The private half of the key is always held in the TPM. The public half can be used to encrypt any piece of data. When it needs to be decrypted, a request is made to apply the private key to the encrypted data. The TPM will only complete the request when the PCRs are in the state defined upon key creation.

2.3.7 The Trusted Software Stack (TSS)

The Trusted Software Stack [211] (TSS) is a specification made by the TCG of support software for operating systems and applications that try to make use of the TPM. It is designed to provide functionality which may not be present on the TPM for reasons of economy, but are essential for services wishing to use it. The TSS is itself split into multiple layers, including the TPM device driver, TSS Device Driver Library (TDDL), TSS Core Services (TCS), TSS Service Provider (TSP) and cryptography services. The TSS is responsible for maintaining the Integrity Measurement Log and swapping encrypted keys in and out of the TPM's limited memory. Several TSS implementations exist. Programs described in this dissertation made use of both the IAIK jTSS [101] and TrouSerS [219] libraries.

2.3.8 Monotonic counters

TPMs must be able to provide at least four *monotonic counters*. Monotonic counters are simple integer values (associated with an identifier) that can be read using the `TPM_ReadCounter` command and incremented via `TPM_IncrementCounter`. Counters can be created and destroyed, but once destroyed, their identifiers may never be used again. There are many suggested uses, including counting the number of times a platform is rebooted, or to implement *count-limited objects* (CLOBs) [186]. Three restrictions apply to counters: it must be possible to increment one continually for 7 years, they must support an increment at least every 5 seconds [212], and only one counter may be incremented on one boot of the platform.

Although counter values cannot be directly attested in the same way as PCRs, it is possible to do an equivalent operation. TPMs support *transport sessions* which encapsulate commands sent to the TPM and provide a signed log of their results [186]. A `TPM_ReadCounter` operation can be called within a session, and the resulting signed log sent to a remote party.

2.3.9 Tick counter

The TPM also contains a *tick counter* which increments steadily over time. This is not a direct representation of the current real time, as it is not required to operate when the platform is powered down. The tick count is begun from the start of a *timing session* which may be the platform boot or TPM initialisation. The platform may then read the number of ticks using the `TPM_GetTicks` method. A protocol for associating tick counts with time can be found in the TPM design principles documentation [212].

In addition to reading the tick counter, it can be used to *time stamp* arbitrary data. The data and current ticks are hashed together and then signed with a TPM key. This can be used to effectively attest the platform's tick count.

2.3.10 TPM Performance

Despite the TPM usually being implemented in hardware, it is not a high-performance device. It is not a cryptographic accelerator and is designed for security and tamper-resistance rather than speed. This means that programs and protocols should use software wherever possible and avoid the overuse of TPM key-based operations such as attestation, encryption and signing. The extend operation is faster, but the overhead of hashing a large file (such as a kernel image) may result in a slight performance penalty. For example, running the `sha1sum` command on a 175MB file took 55 seconds on a Compaq 6510b laptop with an Intel Core 2 Duo processor. More details of the performance impact of the TPM on authenticated boot are given by Sailer et al. [180].

2.3.11 Trusted Network Connect

Trusted Network Connect is a standard proposed by the TCG to specify how network infrastructures should communicate to protect endpoints and prevent the spread of malware [213]. It provides specifications for protocols and functionality to support auditing and access control, based on platform integrity information and user authentication.

2.3.12 Isolation techniques: Virtualization and sandboxing

Isolation mechanisms can be used to separate trusted and untrusted code. This is a useful approach, as it allows the trusted computing base (in the RFC 4949 sense) of a platform to be separate from the rest of the code, but still provide assurance. This reduces the size of the TCB, while still allowing untrusted code to be run. Isolation can be imposed at different levels. Operating systems, in combination with OS paging and rings, provide process-level isolation, as well as isolating the kernel from userspace. Platform-level virtualization, on the other hand, allows for the entire machine to be virtualized. This means that several operating systems can be running, unaware of each other, none of which have sole access to the platform. This form of isolation can be enforced in hardware (through processor features such as Intel Virtualization Technology) or software, or a combination of both. There is a trade-off between

performance and the strength of the isolation provided, a full discussion of which is not within scope of this dissertation. When referring to virtualization, the general architecture shown in Figure 2.2 will be assumed. This consists of a hypervisor at the lowest level, running on real hardware, which then allows several guest virtual machines to run, each completely isolated from the others.

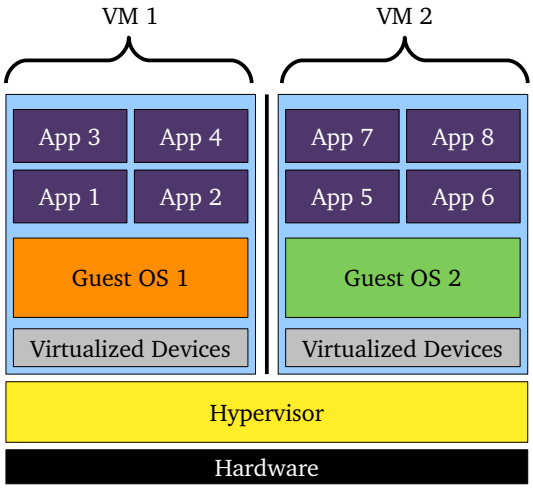

Figure 2.2: The structure of a virtualized platform

A virtual machine can also refer to a language-level runtime environment such as the Java Virtual Machine. This does not virtualize the platform, but provides a sandbox between the executing code and the machine. This is an opportunity for users to impose restrictions on the executing code, such as limiting file and network access. In addition, type safety can be enforced by the VM, as well as exception handling. These eliminate several types of common vulnerabilities. However, the VM itself is complex and may introduce vulnerabilities as well as lowering performance.

2.3.13 Notation

The following notation will be used when describing trusted computing systems and protocols. This is adapted from the syntax of the Logic of Secure Systems (LS^2) [53] and Casper [119].

Cryptographic keys

Cryptographic keys can use any of the following notation.

- Keys can be single characters – e.g. k or j – in which case assumptions about how they will be used by the actors with access to them will be given in the text.
- Public key cryptography involves the use of a public and secret (or 'private') key for an actor, for example: $SK(Bob)$ is Bob's secret key and $PK(Bob)$ is Bob's public key. Actors

like Alice and Bob are assumed to hold their own secret key securely unless otherwise specified.

- Arbitrary key pairs may be shown as the key k and its inverse k^{-1}. The use and assumptions placed on each key will be defined in the text.

Encryption and Signing

Protocols involving encryption and signing will use the following notation from Datta et al. [53].

- $SYMENC_k\{|\ X\ |\}$ is the encryption of value X using symmetric encryption and key k.
- $SIG_{SK(Bob)}\{|\ X\ |\}$ is the signing of value X using Bob's secret key $SK(Bob)$.
- $ENC_{PK(Bob)}\{|\ X\ |\}$ is the signing of value X using Bob's public key $PK(Bob)$.

As in [28], perfect encryption is assumed unless otherwise stated, so that any adversary is incapable of decrypting $SYMENC_k\{|\ X\ |\}$ unless they have access to key k. Perfect public key encryption is also assumed: a message encrypted with a public key can only be decrypted by a secret key, and vice-versa.

Protocols

Protocols will be described using Casper-style notation [119]. A message M from Alice to Bob followed by a reply R from Bob to Alice is shown as:

$$A \rightarrow B : M \tag{2.1}$$
$$B \rightarrow A : R \tag{2.2}$$

Defined terms in protocols

Cryptographic Hash. When used to produce a digest for input X, this can be written as $HASH(X)$, $H(X)$ or using the specific implementation: $SHA1(X)$. Hashes are assumed to have perfect properties: they are not directly reversible, no collisions will occur, and a hash is not commutative, e.g. $SHA1(A \mid B)$ is not equivalent to $SHA1(B \mid A)$, where '|' indicates concatenation.

HMAC a 'Hash-based Message Authentication Code,' a hash of an element, encrypted using a secret key. Notation: $HMAC(k, X)$ indicates a HMAC of element X using key K. The hash function itself may be given as well: $HMAC_{SHA1}(k, X)$.

Nonce a freshly-made random integer, used to establish timeliness. Usually given as just $nonce$ or $nonce_A$ to indicate it was created by user A.

Timestamp a *signed* statement of the current time. These will be given as $timestamp$ or may be shortened to ts_1 and ts_2 to indicate two timestamps, where ts_1 is older than ts_2.

Trusted computing notation

In protocols, the following notation is used to describe TCG objects, keys and parties. Some simplifications have been made from the full TCG specifications.

PCRs. Platform Configuration Registers are show as pcr_x to indicate the content of PCR number x. In some cases a range is given – pcr_{1-9} is every PCR from 1 through to 9 inclusive.

AIKs. Attestation Identity Keys are used in several protocols. The secret half of an AIK pair for platform P is given as $AIK\text{-}SK(P)_1$ and the public half is $AIK\text{-}PK(P)_1$ where the subscript is a label for the particular AIK, as a platform may have several.

Quotes. An attestation takes the form of a TPM Quote. Quotes are written as $Quote_{AIK-SK(P)_1}\{|\ pcr_{1-5}, nonce\ |\}$ where this represents the attestation of PCRs 1-5 signed by $AIK\text{-}SK(P)_1$ with a nonce included. Quotes are actually the hash of these PCRs, rather than PCR values themselves, but this is left off for brevity. Quotes are treated as logically equivalent to signing with an AIK private key: $SIG_{AIK-SK(P)_1}\{|\ pcr_{1-5}, nonce\ |\}$. The shorthand for a platform quote is sometimes used, P_{quote}, but will be defined earlier in the text.

CertifyInfo. The credential for a TPM-bound key, showing that the private half of it is held in the TPM, is described as $CertifyInfo_{AIK-SK(P)_1}\{|\ K^{-1}, [pcr_{x-y}]\ |\}$. This shows that key K is held in platform P's TPM, bound to PCRs x through to y. The credential is signed by the secret half of P's AIK. If no PCRs are specified, then it just certifies that the key is held within the TPM.

Privacy CA. In protocols, the Privacy CA is shown as PCA. When the PCA *signs* something, such as an AIK credential, $SK(PCA)$ is the Privacy CA's private key.

AIK Credentials. The certificate stating the validity of the AIK is given as a credential signed by the Privacy CA: $AIKCredential_{SK(PCA)}\{|\ AIK\text{-}PK(P)_1\ |\}$.

Attestation Parties. The parties involved in an attestation are the *challenger* ('relying party,' 'requester'), who requests an attestation response, and the *responder* ('attester,' 'target platform') who uses a TPM to generate the TPM Quote reply.

Actions

In algorithms, the following syntax is used for certain actions:

PCR Extend. The action of extending a PCR is written as `extend(x, M)`, representing the action of extending item M into pcr_x.

TPM Counter Increment. The action of incrementing a counter with label `label` is given as `increment_counter(label)`.

2.3.14 Summary: Assurance properties through trusted computing

Trusted computing has been designed to enable assurance of several useful properties. As described in this section, use of TPM keys (signing and attestation) can provide unambiguous, re-identification of a platform. One key can only belong to one platform, so seeing a signature by the same key twice provides a strong guarantee of identity. TPM keys can also provide confidentiality through sealing and binding. Data bound to a TPM key can only be read with the cooperation of the platform with the right TPM. Attestation provides evidence of PCR values, which themselves give the ordered sequence of `TPM_Extend` actions performed by a platform. Assuming the boot process follows measure-before-load, and no runtime attacks occur, this might provide a full list of the software running on the platform. Statements such as 'platform X runs Y' can be made, as well as 'platform X has not run known malware Z.' These guarantees are backed by hardware, which makes them stronger than application or OS-level assurance, which might be affected by malware or runtime attacks. However, to go further and make more specific *behavioural* statements, hardware-backed assurance must be combined with further techniques. The following sections provide details on software assurance and isolation, which can provide the missing functionality.

2.4 Specification and Verification Techniques

A more traditional way of establishing whether or not a piece of software will behave properly is through formal verification. This requires a specification of the behaviour of the system, followed by an analysis of the code to see if it conforms. There is a huge range of literature surrounding this topic. A complete survey has not been provided due to the size of the field, but several techniques relevant to the issues arising from remote attestation and web services are discussed in this section.

2.4.1 JML and Design by Contract

The Design by Contract (DbC) approach advocates having a 'precise definition of every module's claim and responsibilities' [136] in order to create reliable and, importantly, *reusable* components. Module interfaces are annotated with pre- and post- conditions in the form of `requires` and `ensures` clauses. There are also class invariants, which express 'general consistency constraints that apply to every class instance as a whole' [136]. Several annotation languages exist for the DbC methodology, including Eiffel, Spec# and JML. JML [112] offers other language features, including specification of exceptions, non-null annotations and class ownership. A simple example of JML can be found in Figure 7.2.

2.4.2 Static program analysis

Static analysis is the process of automatically extracting properties from the source code or binary of an application without executing it. This kind of technique has been used for

a number of purposes, including finding common security problems [41, 40] and spotting memory-management bugs. Static analysis is a broad term that covers simple source-code scanning (perhaps just using a regular expression) as well as more rigorous program analysis with theorem proving.

One such technique is called Extended Static Checking [175]. An ESC tool takes annotated program code, translates it into logical terms, runs the code through a theorem prover with the annotations, and then produces either a counter example or a 'Verified' result. Figure 2.3 is an overview of the process. There have been several extended static checkers developed, including ESC/Modula-3, ESC/Java2 and ESC/Haskell. ESC/Java2 [46] uses JML as the annotation language and can interpret Java 1.4 source code. It is a useful tool for both the Design By Contract and Design For Verification approaches. ESC/Java2 translates source code into predicates and terms which can be understood by Simplify [56], a theorem prover.

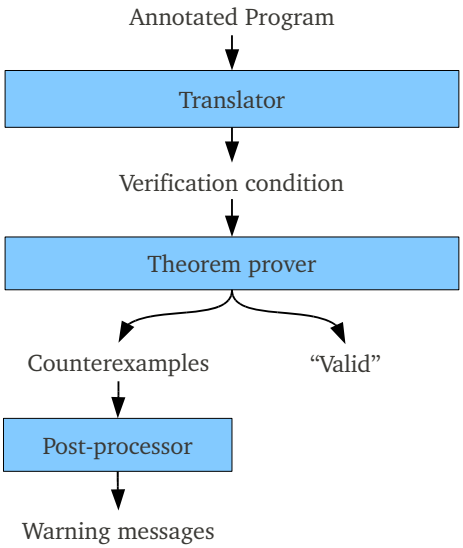

Figure 2.3: The Extended Static Checking process. Figure adapted from Leino [175]

ESC/Java2 has been used successfully in a number of projects and has been used to check software consisting of tens of thousands of lines of code [175]. Rioux and Chalin [172] describe their experiences using it to improve the quality of web applications, as well as providing a good overview of the concepts and terminology. They discovered several faults in their code, including failure to propagate design changes and missing exception conditions. They conclude in favour of the approach, as JML provides a better quality of program documentation and ESC/Java2 makes sure of the accuracy of the annotations. They also believe that ESC/Java2 should scale efficiently. Another example, although this time not using ESC/Java2, is presented by Pavlova et al. [158]. They use JML and JACK, the Java Applet Correctness Kit, for verification. The purpose of their work is to check that smart card implementations meet certain high-level security properties. These include the life-cycle of the card – going into a 'dead' state when misused – transaction atomicity, exception handling and access control.

2.4.3 Proof-Carrying Code

Proof-Carrying Code is an approach for establishing trust in code developed by a third party. The following quote describes the general process:

> 'In a typical instance of PCC, a code receiver establishes a set of safety rules that guarantee safe behaviour of programs, and the code producer creates a formal safety proof that proves, for the untrusted code, adherence to the safety rules. Then, the receiver is able to use a simple and fast proof validator to check, with certainty, that the proof is valid and hence the untrusted code is safe to execute.' [149]

The PCC method has several advantages. Firstly, the code producer does the bulk of the work, creating the safety proof of the application. The end-user just has to run a verifier. Secondly, the code is shipped as a binary, with annotations. Preserving the secrecy of the source code may be important for commercial applications. Thirdly, the system is extremely flexible, with the only trusted code being the final verification program. Proofs may be written by hand or generated by a *Certifying Compiler*. This component can create the proof at compile time.

However, there are several practical issues with using PCC. Creating the proofs is difficult and time consuming. Establishing the safety conditions is also hard, as is expressing all the requirements. Some implementations exist, including an example system by Colby et al. [48] for verifying the type safety of Java applications. Atkey et al. [9] discuss how PCC could be used in a grid computing scenario as an alternative to runtime monitoring of untrusted code. They also present an implementation which checks for conformance with a resource-usage policy. Franz et al. [66] have tried to make PCC more portable and more efficient, by combining it with a minimal virtual machine. This also lets them tailor the VM language for verification.

Proof-Carrying Code is just one of several language-based techniques for adding verifiable information to a program executable. Kozen [108] provides a summary of three other approaches which have a security and safety emphasis. These include the Typed Assembly Language (TAL), Efficient Code Certification (a less-rigorous but faster PCC) and JFlow for information flow properties. Existing research on information flow security is summarised in more detail by Sabelfield and Myers [177] who cover a wide range of language-based techniques.

2.4.4 Booster and model-driven approaches

Booster [54] takes well-defined system models and uses domain assumptions to generate complete object databases. The correctness of the generation process implies a guarantee that the end result is bug-free and potentially trustworthy.

Booster allows the specification of method pre- and post-conditions, along with type information and access control details. The significance is the use of a formal notation (based on Z) to create the service with no manual editing of code or configuration files. It is therefore

sufficient to know that a service was compiled with Booster in order to *guarantee* that it implements the original model. If the generation process is assumed to be correct, then the entire application can be analysed just by reasoning about its specification.

Booster is conceptually a compiler, as it takes a system description and turns it into an executable program. However, this is something of a simplification and it should more accurately be described as a *system generator*. This is because it takes a very high-level model as an input, which by itself contains insufficient information to create an application. This input is then refined by a series of steps to produce the final implementation. At compile time, most of these refinements are *proved* to maintain the same pre-, post-, and invariant conditions that the original model did. The final output is also more complex than the result of a traditional compiler, consisting of SOAP interfaces, a web GUI, the object database and access control mechanisms.

An example usage is to extract data-flow properties from an application. For example, students submitting work to an online system may want a proof that their final mark will never be revealed to fellow students, except in the form of a class average. In a medical record system we could show that information about heart conditions is always visible to the prescribing physician. Such properties can be as fine-grained as necessary or invariant over the whole application.

2.4.5 Specifying and verifying services

There have been many attempts to create better specifications for web services, often in order to improve automatic runtime selection and verification. SOA researchers have developed both OWL-S [128] and SAWSDL [107] which add semantic annotations to WSDL. They can provide pre- and post- conditions and use standard ontologies for describing data types and functionality. They are very flexible, supporting a wide range of rule definition languages within them. However, the main emphasis has been on dynamic discovery and composition of services. Semantics usually refers to the high-level intentions of a service, rather than the specific operational details. This makes the descriptions largely unenforceable, as they have no relation to the code that implements them. These issues have led to recent criticism by Petrie [164] about the impracticality of public service-oriented architectures.

A solution might be to apply formal methods to web service implementations, and there have been many projects aimed at doing exactly this. However, this has mostly been about verifying message interactions. Betin-Can et al. [16, 17] use a design for verification (D4V) approach. They introduce the 'Peer Controller Pattern' for creating reliable services. This separates out the message exchange from the logic, massively simplifying the verification process, which can then be automated. Behavioural interfaces are also generated. Assertions that are known to hold in individual services are then combined using hierarchical interfaces, and the behaviour of the whole system can be checked with regard to synchronisability. Individual services implementations are considered to conform with their interfaces if their call-sequences are acceptable to its state machine. This is verified using JavaPathFinder [230]. This is appropriate when considering concurrency issues, but does depend on all services

being developed by the same people, with trust less of an issue.

Rioux and Chalin [172] use ESC/Java2 to reason about the code of a web-based application framework. They document their experience of using a design by contract methodology in order to assess the effectiveness of ESC/Java2 at increasing program quality. They highlight the difficulty of using a static checker with external libraries which lack a specification. Overall, 90% of faults identified were problems with the inadequacy of the specifications themselves, rather than any coding errors. The other 10% were genuine bugs in the program. Similarly, Heckel and Lohman's also use a design by contract approach (see Section 2.4.6).

Sarna-Sarosta et al. [187] present the idea of using declarative contracts to specify services and then use these to guarantee certain safety properties. Developers would enhance the specification of services with their requirements for exclusive resource access, and then 'containers' compose and negotiate contracts to ensure that all the requirements are satisfied. There would be two types of container: flow and inter-process. Flow containers would address problems of concurrency within a service, whereas inter-process containers work on multiple services, negotiating access to multiple shared resources. This only works when all the services are on the same server, which is also running the container. The main aim of this work is to reduce the overhead on a developer, so that code for synchronising transactions does not need writing, but is automatically created by containers which read the declarative contracts.

2.4.6 Testing services for assurance

Several approaches have been taken to let a requester establish whether a service will work as expected. Heckel and Lohmann [88] use design by contract (see Section 2.4.1) to create web service behavioural contracts, complete with pre- and post- conditions for methods. These contracts are declared in an extension to WSDL. Test cases are automatically derived from contracts and used in order to match the services offered by the provider with the requirements of the requester.

Sharygina and Kröning [189] use model checking techniques to verify that services do not have any concurrency-related problems, such as safety and liveness properties. They define and implement a PHP-like language for web services. Any number of services created with this language can then be checked for safety. One of the important features of their work is that it allows for synchronous communication with other services *and* asynchronous interleaved communication with databases. This work is impressive, but does assume that the verifier of these services is also the designer, and as such is more suitable for testing than remote verification. Furthermore, they readily admit that the application of formal methods is limited, as it would involve having a complete model of all the many library functions within PHP.

Tsai et al. [223] describe a framework ('WebStrar') for web service assurance. Services are registered with WebStrar, which performs a series of tests on it. Each service has an OWL-S [128] specification and this is checked via 'Completeness and Consistency' analysis and model checking of the specification and verification patterns. There is also a step involving positive and negative test cases, which all go towards ranking the services in terms of reliability.

This approach is a logical way of gaining assurance, but does have some flaws. Testing is not appropriate in a situation where the service operates on live data. Secondly, testing is inadequate for demonstrating conformance between a specification and its implementation, although it can raise confidence. It is also not clear whether the tested services have any obligation to re-register in the case of a change to their implementation. This could potentially invalidate all test results.

In summary, most existing work focuses on using verification tools and testing within the development process, rather than as a tool for helping the end user, or enforcing trustworthy behaviour.

2.4.7 SOAP proxies

One method for adding authentication and access control to a web service is the use of a SOAP proxy [25]. These have been discussed in detail by Power et al. [168]. They have the advantage of requiring little or no modification to the base service, while enabling complex policy enforcement.

2.4.8 Summary: Assurance properties through software analysis

Specification and verification can provide assurance (or even proof) of far more detailed, algorithmic properties of software. Static checking can demonstrate that post-conditions and invariants are met, and model-driven approaches can also show this as well as the relationship between data items and objects. Other techniques can demonstrate type safety, the absence of buffer overflows, access control rules and how information will flow. Testing can give weaker, but nonetheless useful evidence of the outcome of a variety of actions, as well as the overall reliability of a complex system.

These properties are generally established through direct, local access to the code or system. However, in combination with the assurances provided by trusted computing and attestation, it seems possible that this could be applied to software running on remote services. This dissertation will look at how practical it is to use attestation for this purpose.

2.5 Conclusion

Several approaches to assurance have been discussed, each with their own strengths and weaknesses. A promising approach is to combine the detailed analysis possible through software verification, with the hardware-rooted assurance provided by TCG-defined attestation. With this in mind, the thesis question – to what extent attestation is a feasible mechanism for gaining assurance in services – can be refined to consider different assurance properties, many of which have been covered in this section.

The rest of this dissertation will consider the practicality of using attestation for the following assurances. Each of these might be considered important for trusting remote services, but offer guarantees of different properties and at varying levels of confidence:

- Unambiguous identification of the service platform.
- Secure communication, without loss of confidentiality or integrity of messages sent and received.
- The absence of known malware on the service (*blacklisting*).
- Only known, trusted software has run on the service (*whitelisting*). Or, similarly, only software from *trusted sources* has run on the service.
- Platform state: a list of all software running, actions it has performed, and each program's runtime memory state.
- Access control policies, whether or not they are being enforced, and any violations.
- Runtime security state: has the platform been attacked, through a vulnerability such as a buffer overflow?
- Service behaviour: what it will do when queried. This may include the range of future actions the service could perform. This is likely to be established through assurance of the service software.

Attestation has already been proposed as a mechanism for most of these properties. The next chapter will look at the well-known problems with doing so, as well as some of the solutions developed in related literature.

Chapter 3

Attestation: Problems and Existing Solutions

The last chapter described the theory behind how TCG-defined attestation works, in terms of protocols and the Trusted Platform Module. In practice, however, this is only one part of a larger process for establishing platform trustworthiness. It turns out that using attestation on real software systems is surprisingly difficult, and the literature is full of criticisms of it [141, 167, 21, 42, 200, 47]. This chapter lists these problems and presents the various solutions that have been developed in the last decade of research. In particular, the different platform-level *integrity measurement* strategies are covered in Section 3.3. In Section 3.4, the current state-of-the-art is analysed to identify the areas which are still outstanding.

This chapter contains an analysis of the key challenge that this dissertation aims to solve, and the contributions in the following chapters all seek to overcome the problems listed below.

3.1 Open Problems

There are 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*). In addition, there are the problems of the number of credentials and trusted parties required (*trusted parties*), the *performance* impact, application *compatibility*, establishing a *trusted path*, and attestation across *multiple domains*.

3.1.1 The semantic gap: Measured but not trustworthy?

Attestation has been criticised for reporting a platform's *execution state* rather than its *security state* [167], which many consider to be the ultimate goal. These two properties are related, but there is a significant gap between them. If it is not clear that one software configuration

is necessarily more secure than another, why report it? The root of the semantic gap problem is that integrity measurement only provides assurance of the identity of software loaded, and additional software assurance methods are required to make further security guarantees. However, if the only assurance goal sought is to identify malware, for example, then arguably attestation is more appropriate. Assuming measure-before-load is implemented correctly, *blacklisting* is a simple matter of identifying the components in a list that are known to be untrustworthy. Unfortunately, this is rarely the case, and assuming measure-before-load is sometimes unrealistic.

Sadeghi and Stüble [178] introduced 'Property-Based Attestation' (PBA) to solve the semantic gap problem (see Section 3.2.1) but still rely on at least one party being able to match software identity to security properties. Presumably this would be achieved through testing or verification, both time-consuming processes. Even more complexity is apparent when the scope of measure objects is expanded beyond just executables. Platform configurations include configuration files, data and runtime events, all of which might need reporting in order to gain a thorough impression of the state of the platform.

Arguably more difficult than reporting security state is reporting how a platform will *behave*. This is essential if attestation is to be used for establishing *trustworthiness*, as the two concepts are linked by the TCG definition (see Section 1.3).

Some solutions do exist. Semantic Remote Attestation (discussed in Section 3.2.4) attempts to bridge this gap through reporting dynamic runtime information. Similarly, the Tisa system by Rajan and Hosamani [170] allows for reporting of program execution traces, through a trusted monitor which instruments Java bytecode. Requirements can then be specified in terms of linear temporal logic expressions. However, this relies on the program user understanding the meaning of code execution traces, and on the correct implementation of the code, monitor and middleware. Furthermore, the configuration of the monitor will be complex and will affect trustworthiness. An alternative approach, taken by Alam et al. [2, 3, 4], is to use a trusted virtual machine to log behavioural updates to objects, in order to link attestation to usage controls. This relies heavily on the implementation of the virtual machine, and has only been used to investigate usage control issues on client machines. Another attempt to link attestation to security state is through use of vulnerability databases. Munetoh et al. [142] report whether or not any executables have known exploits as listed on the Common Vulnerability and Exposures [138] database. Unfortunately, this approach is limited to identifying existing flaws, rather than pro-actively defending against new ones, and relies on the accuracy of the database. St. Clair et al. [198] propose to link the definition of a trusted platform state to the original software installation, a principle similar to the 'birth certificates' proposed by England [61]. They use a custom installer to set-up and configure a platform, and thus create the 'known-good' image. Any deviation from this image is considered untrustworthy. Coker et al. [47] agree that the semantic gap is a problem, and their fourth principle for an attestable system is that the semantic content of attestations should be explicit. They state that an appraiser should be able to infer consequences from a series of attestations. Their proposed architecture has an *Attestation Manager* component, which is responsible for using a suitable

tool to analyse attestations, before then passing the analysis on to the challenger in a standard format.

3.1.2 Vulnerability to runtime attack

Integrity measurement can assert the identity of software when it was originally loaded, but says nothing about the *runtime* state of the platform [188, 61]. 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. This problem is directly linked to that of *semantics*, as the trustworthiness of the platform is dependent on both static binary state and runtime. Therefore, many proposals attempt to solve both problems at once.

One approach is to monitor a platform at runtime, using a trusted agent. Kil et al. [105] augment the operating system to monitor system calls and applications, and extend a PCR when any bad behaviour is observed. This is linked to program semantics through earlier static analysis of the applications, identifying common patterns of execution. Unfortunately, this approach does not detect all integrity violations, and has a performance overhead. Gu et al. [83] measure all dependencies and inputs to the application being monitored, through trapping system calls. Zhang and Wang [242] propose to attest to process trees, rather than just binaries, to detect when unexpected processes are spawned, highlighting an attack. Baiardi et al. [10] use virtual machine introspection to monitor the platform, allowing flexible auditing policies to be followed. These approaches undoubtedly make it harder for a runtime attack to go unnoticed, but all make the challenger's task more difficult. The level of knowledge required for the challenger is much higher, as they must understand the implications of any policy violations. At best, these proposals fight the symptoms of the problem, rather than immunising against the cause – excessive TCB size and complexity.

Unfortunately, runtime attacks compromise almost all the assurance goals described in Section 2.5. Reliable blacklisting and whitelisting is not possible, as a remote attack could instantiate any software. Any reports of platform state could be compromised, and behaviour could be changed in almost any way.

3.1.3 Maintaining a whitelist

The complexity of managing a large software whitelist has frequently been cited as a major problem for attestation. England [61] claims that the 4 million windows drivers (growing at 4000 per day) makes even identifying the software running on a platform a challenge. Other researchers have made similar points about the number of possible configurations [178, 85, 173]. However, there are some promising counter-examples. Sailer et al. [179] show an implemented network access control system which uses a whitelist of only 25000 entries, and is designed to handle application updates. An enormous amount of literature exists on platform minimisation and TCB reduction. Attestation highlights the problem well, as measurement numbers are a quick metric for comparison. The whitelist problem is clearly related to the runtime issue, itself a part of the *semantic gap* issue. TCB minimisation is therefore crucial to making attestation

feasible, as it contributes to all of these problems.

The most obvious approach to TCB minimisation is to use a small operating system and software stack. Böttcher et al. [20] use the L4 microkernel to do this. Indeed, a smaller version of L4 has even been formally verified [106], enhancing assurance of any system using it. Singaravelu et al. [194] reduce the TCB further through 'AppCores,' into which the security-sensitive portion of applications are placed. This means that the rest of the application can remain untrusted. However, attestation of this process is not considered. The LibraryOS project [7] also allows the creation of minimal application environments, useful for security-critical components. Another technique is to use information flow controls to reduce the number of applications that need measuring. Jaeger et al. [96] use SELinux to do this, allowing untrusted applications to be ignored and not measured. However, the overhead in policy is significant. The IAIK Privacy CA project [151] has another approach to reducing the TCB. They analyse the precise components in the Java runtime that are actually used, and remove everything else before deployment. There is an enormous amount of literature related to TCB reduction, and a full review is not presented here.

One alternative is to measure a virtual machine image rather than an entire software stack, turning hundreds of measurements into just one. The implications are discussed in detail in Section 3.3.2. Toegl and Podesser [218] propose per-application VMs in part for this reason, and achieve integrity measurement logs of only twenty or so entries. Wang and Wang propose the same for VMs in a grid system [231], and England [61] also suggests attesting VM images with known security properties. Cooper [51] has an alternative approach to minimise a grid platform. He proposes that only a 'job security manager' should be attested, which then implements security controls, such as providing encrypted storage and isolated job execution. This works for grid jobs, but would be difficult to implement for web services.

The *software update problem* is referred to as one cause of the whitelisting problem. Frequent software patches make the whitelist too large and dynamic. Property-based attestation can help with this issue, as can the chameleon hashing approach taken by Alsouri et al. [5]. Chameleon hashes use a key-based hashing function to allow different files to produce the same hash value, thus avoiding increasing the size of the measurement list. However, this also prevents revocation of software when a new vulnerability is found. England [61] suggests that a 'birth certificate' should be used instead, an attestation of the original installation image, rather than the running, patched version. This is a clear trade-off between detail and manageability.

Configuration files are also part of the issue, as these are likely to be unique for every platform, despite potentially having the same meaning. St. Clair et al. [198] avoid this by generating configuration during a measured installation process. Alternatively, the SAConf proposal [232] uses a configuration analyser to link this to file semantics, rather than binary identity. This is discussed further in Section 7.6.

Using a dynamic root of trust can also reduce integrity measurements significantly. McCune et al. [130] use it to launch isolated pieces of code despite the presence of an untrustworthy operating system. However, the size and complexity of the code is naturally limited, as none of the operating system provided services are available. The Oslo bootloader also uses the

DRTM to remove BIOS and pre-boot measurements [103].

3.1.4 Too many trusted parties and processes

Bottoni et al. [21] summarise the credentials and beliefs necessary in order to use remote attestation. The results are not encouraging. For a simple scenario, where the remote platform has three software layers and there are two Certificate Authorities, five authorities must be trusted and *fifteen* certificates verified. Furthermore, assuming that the processing of certificates will rely upon checking revocation lists, remote attestation becomes liable to blocking and denial of service. However, many of these certificates might well be provided by the same party (e.g. a system administrator) and as such the complexity is diminished significantly. The key principle to take from this problem is that adding new trusted third parties should be avoided when designing attestation-based systems.

3.1.5 Privacy concerns

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 [178, 167], requiring software from only one vendor, for example. This could work against the user's best interests. Furthermore, reporting the exact hash values could make an attacker's job easier [110], as he or she will be able to quickly identify which known exploits are appropriate.

The problem of preserving privacy is closely related to that of whitelisting. If precise integrity values do not need to be disclosed, then a precise whitelist also does not need to be maintained. As a result, many of the solutions discussed in Section 3.1.3 help with privacy too — in particular, property-based attestation and chameleon hashing [5]. Another approach is to use a higher level of abstraction during attestation. Nagarajan et al. [145] put low-level components into 'buckets' and then have multiple layers of properties, which are fulfilled by having at least one component in the right bucket. This theoretically enables the attestation of just the platform-level properties, so flexibility in individual components can be maintained. However, it assumes agreement on the component-property mappings, the transitivity of properties, and has not been shown to scale in a real scenario.

3.1.6 Performance

Because trusted computing features depend on use of the TPM, they are also constrained by the speed in which the TPM can perform encryption and signing. If attempting to establish trust in a remote server, regular attestation or sealing could impact the processing time for requests. This could result in reduced service or an availability issue.

While the TPM itself may increase in performance, there are other solutions. Stumpf et al. [200] propose three improvements on attestation. Firstly, by batching attestation requests together, one attestation can serve multiple remote users. The second technique uses a trusted

third party to regularly request an attestation at fixed time intervals. The attestation result and nonce are published by the trusted party. This guarantees freshness within the time period, and saves individual users from having to request attestations. The last method uses the TPM's tick counter, details of which can be found in the paper. An alternative suggested by Löhr et al. [116] is that certified PCR-bound keys would be more efficient for attestation, compared to TPM Quotes, as these can be used offline by the remote party.

3.1.7 Compatibility with legacy systems

All applications on the attesting platform must support *measure-before-load* for any data they will execute. This means that virtual machines, programs with plug-in architectures and programs with detailed configuration settings must all be modified. This requires a significant amount of time and effort from all developers. The quality of the applications is also important, as any error in the implementation might allow an executable to be loaded without measurement. This issue has occurred in some of the early trusted bootloaders [167], and can undermine many assumptions necessary for assurance.

Solutions to this problem often involve policies and OS-level instrumentation, such as the IMA system [180]. These allow unmodified applications, but have an overhead on policy, and lack the intelligence to distinguish some executables and static files. Alternatively, Kil et al. [105] use static analysis of legacy executables as well as OS-level system call tracing to enable better monitoring. Dietrich et al. [58] propose an architecture for legacy systems which provides attested communication channels through *attestation proxies*. However, this does not help with the integrity measurement process itself, but can benefit applications which are working on compatible systems. The TCG solution, on the other hand, is to use the Platform Trust Service (PTS) to enable monitoring of the whole system. This is equivalent to an operating-system level solution, as it is also placed in the TCB. More discussion of the PTS can be found in Section 6.6.6.

3.1.8 Establishing a trusted path

Remote attestation preserves the privacy of individual platforms by introducing a pseudonymous attestation identity key (see Section 2.3.3). However, AIKs are not meant to be used for anything except attestation, and cannot be used in further protocols. This means that it is difficult to establish that the attested platform is the same as the platform being communicated with, as nothing links the AIK with a particular transport session. A man-in-the-middle could forward valid attestations from another platform to convince the relying party of its trustworthiness.

This problem of establishing secure channels to trusted platforms has been discussed extensively [42, 74, 72, 201]. The solution presented by Goldman et al. [74] links a platform's SSL key to its AIK, which then makes it easy to establish a transport session with the attested platform. They do this through a number of mechanisms, but one is to measure the public SSL key at boot time, so that all attestations must include it. Choi et al. [42] point out that this does

not work if a malicious platform manages to get the SSL key, as they can then use a 'good' platform to attest but switch to a 'bad' platform with the same key afterwards. Their solution uses a network monitoring agent and trusted third party to guarantee the endpoint address of the attested platform. Stumpf et al. [201] describe a masquerading attack on standard attestation, and present a robust integrity reporting protocol as an alternative.

3.1.9 Attestation across multiple domains

Another problem with attestation is the application of integrity measurement policies across multiple administrative domains, such as in grid systems. If certain software configurations are unique to one domain, then platforms in these domains will not be trusted by other domains, as the attested configurations will be difficult to validate [92].

3.2 Related Research, Systems and Tools

In this section a few of the most significant research contributions will be discussed in further detail, to identify the principles behind them, and any areas for improvement. Many projects involving attestation have encountered some of these problems before, and are analysed to see how they have been overcome.

3.2.1 Property-based Attestation (PBA)

The property-based attestation approach [178] proposes that platforms should attest to properties of the software they are using, rather than just hash-based identities. This reduces privacy concerns (exact configurations do not need to be revealed [36]) and software updates become unimportant, so long as the same properties are maintained. Multiple vendors can produce software which has the same property, avoiding any potential for vendor lock-in. Property-based sealing is also attractive, as all trustworthy configurations retain access to sealed data without re-sealing on every update. Poritz [167] claims that normal attestation *is* a form of PBA, however, but with a very simple model. The only property being attested is that software with a certain hash value was run in a certain order at system boot. He calls this 'Binary Attestation.' The main research challenge is to implement PBA in a secure, simple, low-infrastructure manner.

Sadeghi and Stüble [178] give a range of implementation options. The basic suggestion is to have a trusted third party providing a layer of indirection between properties and PCR values. The TTP would issue a certificate stating that certain PCR configurations correspond to a certain property. Alternatively, methods involving zero-knowledge proofs and proof-of-membership protocols are discussed which do not require any additional trusted party. This idea is fully realised through a protocol by Chen et al. [35] and then with ring signatures [36]. Each of the approaches has advantages and disadvantages, discussed fully in the paper. However, at no point is the question of property-extraction answered: how should certain properties be

established in the first place? This is a problem, as it is very difficult to establish the behaviour of any piece of software.

3.2.2 IMA and PRIMA

Sailer et al. [180] introduce the Linux Integrity Measurement Architecture (IMA). They tackle several practical problems with TCG technology, including how to measure modules and programs loaded in a seemingly random, non-deterministic way on top of the operating system, while still being able to report the system state in a meaningful way. This is difficult to do when using a simple chain-of-trust, as a different order of program execution will result in a completely different final hash value. In their solution, the Linux kernel measures and extends programs and libraries into the PCRs and keeps its own in-kernel list. This list can then be checked against the PCR value in the TPM. In order to improve performance, measurement results are cached and files are only re-measured when they change. They step through an example web server, running Tomcat and Java servlets, showing which parts of the system require measurement (as they can affect the system) and which parts do not. Several problems are identified here. Knowing precisely which files are actually used by an application is difficult, as many programs can load multiple configuration files from arbitrary locations. Furthermore, dynamic data cannot be measured in the same way as code, so security policies and data histories must be relied upon instead. The authors maintain that their approach is practical, as a 'normal' RedHat 9 Linux system used for writing papers, compiling programs and browsing the web accumulates no more than 500 measurement entries. This claim is based on an old version of IMA. More recent versions measure more components, and have policies associated with them. Section 4.2.4 has more up-to-date statistics. However, the integrity measurements are in no way linked to platform behaviour, and the authors make no attempt to convert them into a trustworthiness value of any kind.

The IMA implementation is improved by Jaeger et al. [96]. They attempt to show the CW-Lite integrity property of a system, a slightly weaker version of the Clark-Wilson integrity model, which requires that high-integrity processes accepting low-integrity data need to have interfaces with filters. These filters are trusted to discard or upgrade low-integrity data inputs. This is shown through measurement and attestation of an SELinux policy which enforces information flow. It guarantees that high-integrity software only receives input from high integrity sources, or from low-integrity sources which are filtered. As a result, any untrusted software or data does not need measuring because the policy prevents it from communicating with high-integrity components. PRIMA was designed to show that a *behavioural* property could be enforced through trusted computing, rather than just secure boot. This has been demonstrated to some extent, although it is impossible to be sure that no covert channels are present. The second aim was to reduce the size of the TCB and therefore the number of necessary integrity measurements. This does seem to be the case, although with the cost of measuring a massive (and complex) security policy of over 1MB. No details are given as to the expected size of the new TCB. It is also difficult to judge whether the added complexity of this system is justifiable, as opposed to maintaining stronger isolation between low-integrity data

and high-integrity, for example with two physical machines.

There have been two similar approaches to PRIMA. Sandhu and Zhang [181] measure the OS up to a trusted reference monitor (TRM). The TRM enforces an access control policy to create protected runtime environments for each application. This gives each application a protected memory space and also controls any secure channels they may want to establish to other pieces of software or I/O. The TRM securely holds an asymmetric key pair for every application. The TRM is capable of enforcing usage control constraints, such as limiting the number of times a file can be viewed, or the application which can be used to view it. Marchesini et al. [126] also measure the system up to the kernel, and then the *Enforcer* software module takes over. This maintains a 'long-lived' core kernel, which checks that a signed, up-to-date *Security Admin* is present. This is responsible for holding a signed description of the 'medium-lived' software on the platform. The Enforcer makes sure that the description of software in the Security Admin matches the current system. All encryption keys for sensitive data are controlled by the Enforcer, which can restrict access to them should the current system not match the expectations. Both of these papers rely upon sensible matching of program identity to trustworthiness, and can only enforcing very limited policies, in terms of access control or confidentiality. More sophisticated statements about behaviour are not dealt with.

3.2.3 Attested Append-only Memory (A2M)

Append-only memory is an abstraction designed to provide a *trusted log* [43], a secure history of events. This provides a mechanism for implementing protocols 'immune to equivocation,' so that one platform cannot lie in different ways to different parties. With A2M, platforms are 'forced to commit to a single, monotonically increasing sequence of operations' [43]. This would be ideal for reporting an electronic ballot, or an audit log. Levin et al. [113] implement a similar system using a single trusted counter and a key, and demonstrate that the functionality can be provided by the Trusted Platform Module.

The use of secure coprocessors have also been suggested for enhancing the trustworthiness of electronic auctions [162, 11], a related problem.

3.2.4 Semantic Remote Attestation

Haldar et al. [84] begin to fill the gap left by PBA, i.e. the mapping between software and behaviour. They use a Trusted Virtual Machine to attest to high-level properties of the running code. The presence of the TVM is attested first using normal methods. The properties that can be extracted and proven by the TVM include class hierarchies, Java VM security constraints and runtime dynamic state. Going a step further, arbitrary properties can be proven by requesting the TVM accept and run code written by the attestation requester. This might check for any kind of runtime or code property. A test suite could be sent to the attesting platform which checks its floating point precision, for example. The integrity of these results are guaranteed by the presence of the TVM. This approach is extremely flexible and allows attestation of meaningful information as opposed to merely program identities.

However, there are some concerns. Firstly, a TVM is a significantly large element in a trusted computing base, and the fact that it must be run constantly makes it a target for remote attacks. It also imposes a performance penalty which may be unacceptable. Moreover, remote platforms will be unwilling to run the arbitrary test code sent from a potentially untrustworthy source, due to security concerns, especially when it can (at best) only demonstrate that their platform *might* be secure. Finally, these tests will presumably need to be done regularly, to make sure that code has not changed in-between. This is a large overhead for the attestation requester.

3.2.5 Model-based Behavioural Attestation

Alam et al. [4] and Nauman et al. [147] propose Model-based Behavioural Attestation. They also identify the semantic gap problem with attestation, and aim to solve it with a trusted virtual machine, which logs behavioural updates to objects in order to enforce usage control constraints. This has also been implemented for web services [3]. One of the most significant contributions is the formal model and overall framework they discuss, which is independent of the attestation technique. However, much of this work focuses on problems of usage control for client platforms.

3.2.6 UCLinux

UCLinux [111] is a Linux security module designed to provide a usage-control system. In doing so, it introduces several useful ideas to mitigate problems with authenticated boot and TPM sealing. These include *TCB pre-logging*, which pre-measures all potential applications so that PCR state does not change while the platform is in use. They note that user login may be an issue, and propose to extend PCR values on this event, to drop any open security contexts. However, recovering from this requires a reboot. They also count their integrity measurement log (to provide a statistic for the *whitelist* problem) and find a total of 419 components in the TCB.

3.2.7 Virtual machine introspection

Virtual Machine Introspection [70] is a way of monitoring a 'guest' virtual machine instance by allowing another virtual machine to run in parallel and inspect its memory and system state, usually for the purpose of intrusion detection. This architecture has the advantage of isolating the inspection VM against a compromised guest, as well as placing it on the same hardware, giving it sufficient visibility for accurate monitoring. Of course, it may still be possible for an intruder to work around the inspection VM. The inspection mechanisms might still be exploited, as they must read the guest VM's memory, over which the attacker can gain complete control. Alternatively, the privileged virtual machine monitor could be attacked. However, the difficulty of such exploits should be significantly higher than any system which does not use hardware isolation.

Baiardi et al. [10] have used VM introspection to provide semantic attestations. When attempting to gain assurance of a remote platform, users can communicate with the introspection VM which can report more detailed state information about it. Trust is established by first challenging the introspection VM to attest its configuration in the normal, TCG-defined manner.

3.2.8 Terra

Garfinkel et al. [71] propose Terra, a trusted computing architecture supporting attestation and virtual machines. This is one of the pioneering papers in the field, and many of the proposals have been adopted by the Trusted Computing Group since.

They extend the chain of trust from hardware to a trusted virtual machine monitor, and then to the VM and applications. This full chain can be attested to a remote party. The problem of software whitelisting and updates is identified. A proposed solution is that the attesting platform provide all the necessary certificates to the challenger, covering each version of the software. Software updates are handled by simply downloading a new version of the certificate. This is similar to the TCG mechanism of having reference integrity measurements, but relies on challengers being able to verify certificates, and having a good revocation strategy.

3.2.9 The middleware problem

Cooper [51] describes the 'middleware problem' for grid security, an issue that is also directly relevant to the trustworthiness (and attestability) of service-oriented architectures. He argues that middleware is highly likely to be the cause of vulnerabilities, as it is large and contains privileged code. It also stores credentials, and is in charge of authentication and access control. The example given by Cooper is of the Globus Toolkit, but the same principles apply to web service middleware such as Glassfish. Glassfish has over 700,000 lines of Java source code, as well as 11,000 lines of C and 66,000 lines of XML. This amount of code will dwarf anything else on a platform, apart from the operating system itself.

3.2.10 E-Voting

Sandler and Wallach [182] suggest using attestation and TPM counters in order to create high-integrity logs of electronic voting systems. The requirements they attempt to fulfil in VoteBox [183] are similar to those discussed in Section A.2. Böttcher [20] suggest using attestations for an anonymity service, to demonstrate that no additional logging component has been installed. They also reduce the TCB of the system, using the L4 Fiasco microkernel operating system and the OSLO bootloader (see Section 2.3.5). In order to establish the integrity of voting machine software, Gardner et al. [68] suggest using the Pioneer system. This can present users with a 65-bit checksum, demonstrating that the right software has been loaded.

3.2.11 CA-In-A-Box

Franklin et al. [65] describe their experiences in creating an attestable certificate authority (CA). Their goals were (amongst other things) to make the CA verifiable to a remote party, and to enforce that only a properly configured platform has access to its signing key. In their solution, they split the system state, so that configuration files are signed and stored on a USB disk and the executables and signing key are stored on the hard drive. The key itself is *sealed* and can only be accessed when the CA boots the correct binary kernel image. Overall, their system is an excellent example of how a trustworthy service might be implemented, and is the first demonstration (to an extent) of the feasibility and costs associated with doing so.

3.2.12 Trusted Grid Architecture

Löhr et al. [116] describe their Trusted Grid Architecture (TGA) which proposes a 'scalable offline attestation protocol' to make sure that grid provider systems are in a trustworthy configuration. They overcome the issue of performance by using a *sealed-key* approach, where messages are sent to grid providers encrypted with a TPM key. This key is sealed to certain pre-defined PCR values, making the data inaccessible otherwise. This method inspired some of the solutions proposed in Section 5.2.1. However, their approach relies heavily on sophisticated middleware, an issue which does nothing to help with the middleware problem discussed in Section 3.2.9. They also identify that this system suffers from problems with software updates and privacy.

3.2.13 Integrity measurement for the Android platform

Nauman et al. [148] describe their implementation of integrity measurement on the Android mobile platform. They compare two methods – attestation of applications and attestation of individual class files. There is a clear trade-off, as class-level measurement is more flexible, but requires a measurement log of around 1941 entries, whereas only an average of 28 application-level measurements are required. They also address the problem of measuring classes loaded across a network, and break down the classes themselves into small components to improve performance and remove redundant information. Classes are divided into meta-information (class name, class loader, descriptor, etc.), 'passive entities' such as static fields and method names, and executable code. However, they do not present a method for verifying integrity measurements, and have not considered how to connect the classes being measured to any notion of *assurance*, beyond basic whitelisting.

3.2.14 Flicker and TrustVisor

As described in Section 2.3.5, one way of reducing the *whitelisting* problem is to use a *dynamic root of trust for measurement*. This was designed primarily for the launching of a trusted virtual machine on an already-booted untrusted operating system. However, McCune et al. [131] use it in the Flicker system for a different purpose: the measurement and attestation of small

elements of critical code known as 'Pieces of Application Logic' (PALs). These are late launched, which pauses the running untrusted operating system, executes the PAL in isolation and then resumes the operating system. Because the DRTM is used, only the PAL and supporting code (the Secure Loader Block or SLB, equivalent to the MLE on Intel platforms) needs to be measured and trusted, resulting in few integrity measurements and a small chain of trust. McCune et al. suggest that their technique could be used in many scenarios, including the following four: for running a trusted rootkit-detector on client platforms, to execute a simple distributed-computing application (such as factoring a large number) in a verifiable way, to protect SSH passwords on untrusted remote machines, and to protect a certificate authority's private signing key. Because of performance constraints, Flicker is most appropriate for relatively infrequent events (such as password entry). Another constraint is that because the existing operating system is unavailable to the PAL, and the desire is to keep the TCB small, the PAL must be relatively simple and not rely on additional libraries or a runtime environment. It must also not use hardware interrupts, reducing possible functionality greatly. The Flicker approach is clearly a useful technique, but not appropriate for the attestation of a complete web service, where the trusted code (based on the definition in Section 1.3.2) is significantly larger. It would not make sense to put an entire web service into a PAL, as the dependencies and support code would quickly make the improvement in TCB minimal.

The TrustVisor [129] hypervisor takes advantage of the earlier Flicker architecture to provide execution integrity for security-sensitive portions of applications. The main focus is on overcoming the performance penalties associated with late launch and the TPM and therefore making PALs more practical. This is achieved by providing a micro-TPM in every PAL which replicates the low-performance TPM operations. Again, PALs are constrained as they are not able to make system calls and must be self-contained. TrustVisor is therefore a more practical implementation of Flicker which can provide enhancements of the security of small amounts of code in otherwise untrusted and unmodified systems.

3.3 Integrity Measurement Approaches

This section looks at the different strategies for integrity measurement, with respect to the existing literature, to see how the problems with attestation identified in Section 3.1 are affected. In particular, there have been several attempts to minimize the number of integrity measurements through either measuring a larger component (a virtual machine) or automating the validation process so that individual hashes do not need to be saved on a whitelist. However, not all strategies are appropriate for all parts of the system, so this analysis begins with a taxonomy of measurable components.

3.3.1 A taxonomy of measurable components

The following components may require measurement and reporting:

Hardware. Devices such as the CPU, motherboard, and network cards. Measurement of these

is pre-defined.

BIOS and firmware. Software loaded onto hardware. Rarely updated.

Boot-time components. This includes the bootloader, kernel image, kernel modules, and any hypervisors or VMMs. These are typically highly privileged, and may implement security controls. Often the boot-order is fixed, and little user interaction (beyond selecting from a menu, or escaping into a different mode) is expected.

Services and daemons. Running in the background, daemons such as cron and SSHD begin without any user input, and run for a long time. This makes them vulnerable to runtime exploits. With normal measurement approaches, only their *start-up* is measured, meaning that challengers must assume that attestations containing these executables are still running them. Web services are also included in this category.

Interpreters. Programs such as the PHP interpreter. They may not be as highly privileged as the OS itself, but they are also responsible for enforcing access controls and loading a large amount of code, including potentially untrustworthy applications. Interpreters must have their own integrity measurement implementation, as they will load files that may appear to be data (e.g. bytecode) to the operating system.

User executables. Applications typically run at the user level, such as an email client or browser. They may be started and stopped, and are often not part of the TCB of the system.

Shared libraries. Part of standard applications, but an incorrect implementation may have a larger impact. More likely to be part of a TCB, as they may be used by services or daemons.

Scripts Bash scripts, python, perl, etc. They differ from executables in that they are frequently user-generated or modified.

Command-line arguments and environment variables. Similar to scripts and user executables, the command line arguments passed to an application often change the expected behaviour.

Configuration files. Files designed to intentionally control application behaviour, as opposed to general data. Sometimes these will be in the form of scripts.

Data. Any non-executed file. However, many data files contain scripts (spreadsheet macros, for example) and have executable content.

Not all of these require alternative strategies to measure, but a comprehensive integrity measurement approach must be able to deal with them all. Because there is some overlap and non-exclusivity of the terms, it makes sense to think of this more as a sliding scale, with privileged, essential components at the top and dynamic, but not frequently executed files at the bottom. Furthermore, it might be assumed that the rate of change (volatility) of the

components is likely to be similar, going from slow at the top to fast at the bottom. However, in Section 4.2.3 this assumption is challenged. Hardware is an exception in this list, as it can only be measured and handled in pre-defined ways.

There are other measurable items which break from the general pattern on this list as they are *runtime* properties rather than load-time. Firstly, significant runtime *events* or actions can be measured. For example, the operating system might measure a user login, or certain system calls [83]. An application could add a measurement when it connects to a remote server. This would be done to preserve a history of the action, perhaps to enforce a temporal constraint [111, 147]. Secondly, an active research area is how to measure and report the *runtime state* of a platform, as opposed to its load-time state. There are many strategies for doing this, some of which involve inspecting memory [70] or data structures [118]. Perhaps the most important missing item is *user input*: what has the user done on the platform? Sometimes this will not be important — the user is treated as potentially malicious and untrusted. On other occasions, such as when the user is a system administrator, there may be semi-trusted things that the user might do which would just *reduce* the trustworthiness of the platform. In these situations it may be worthwhile to record and attest to some user actions.

3.3.2 Granularity of system attestation

There are several levels at which software integrity can be measured. One consideration is the *size* of each item that will be measured. Several approaches have been proposed: measurement of individual applications, measurement of virtual machine images, measurement of application-level events, and hybrid schemes. This can also be considered the stage in which measurement *stops*. In this section the advantages and disadvantages of these are analysed, with respect to the problems discussed earlier.

Virtual machine measurement

A virtual machine encapsulates all the software and state of an individual platform, bar the hypervisor, and therefore seems an important component to measure. A hypervisor could be configured to measure the virtual machine image, typically only one file, and then ignore the individual applications run within it [231].

The first advantage is that only one item needs to be measured, reducing the size of a corresponding whitelist. This has good implications for sealing, as any items sealed to this VM will always be available, no matter the order in which programs are started. Furthermore, individual applications do not need to support measure-before-load, and users are given the freedom to use any applications provided by the virtual machine that they want. The VM measurement can also be considered reliable, as it only depends on the hypervisor. Performance may be better as only one hash is needed, reducing the impact on runtime applications.

However, this broader measurement only increases the semantic gap between execution state and trustworthiness. The VM image can only describe the range of possible applications that may be run and the operating system. This does not include details on precisely which

programs are run. The OS must further be relied upon to prevent arbitrary applications being downloaded and executed. In essence, VM attestation is only as useful as the operating system's policy enforcement and specification. Furthermore, assuming a lenient policy, the amount of potentially-running software may be larger (any program in the VM), and this actually *increases* the burden on the challenger. They must now be confident that runtime attacks do not exist in any of these components, not just the ones in use. Furthermore, if the VM is to be updated, there will still be significant effort required by the challenger to make sure the right image is kept in their whitelist. This implies a closer link between the VM provider and challenger.

Application measurement

The IMA [180] system measures the operating system kernel modules and every application upon first execution. This approach has already been analysed in previous sections. The main advantages are that only the applications *in use* need to be trusted by the challenger, and that upgrades only invalidate the whitelist entries of the individual upgraded programs. Disadvantages include lower privacy and the impracticality of sealing to system state. Whitelists must also be much larger, although the sources of hash values can be the vendors themselves. There is still a semantic gap to be overcome, and the sequential, measured chain of applications is at odds with the parallel, multi-tasking nature of operating systems. This problem is a major motivation for the work presented in Chapter 6. Furthermore, the operating system must be trusted to perform and track measurements, which may be unrealistic given their large code-base and vulnerability to attack.

Event measurement

More fine-grained than application measurement is the idea of measuring *events* that occur at the application and operating system level. These may include key state changes, such as a user logging into the system, or an access control decision being made. This is similar to Semantic Remote Attestation [85] and Behavioural Attestation [4]. More details can be found in Section 3.3.3.

The clear advantage is that event measurement should be a much closer representation of actual system behaviour and state. Use of PCRs makes the events append-only, and so this provides a greater level of assurance compared to simply analysing the logs of an application. In addition, event reporting – when implemented well – should be more feasible than full behavioural verification and testing of applications, which are difficult for large code bases. Event triggers can be programmed in a semi-automated manner (e.g. by wrapping certain system calls [83]). They might also detect runtime exploits, as an unusual sequence of events could indicate compromise.

However, event reporting requires greater modification of executables to enable compatibility, and will have a performance impact, as they would need to use the TPM regularly. Verification of the measurements will become much more complex, requiring a larger, multi-

level whitelist, as well as an understanding of the applications at a code-level. Privacy is also affected. The trusted computing base of such a platform also becomes the entire set of programs with access to the TPM, and full isolation is required to avoid measurement of arbitrary events by malicious applications.

Hybrid schemes

None of the three levels described in this section are mutually exclusive, and various hybrid scheme are possible.

One option is to measure at the VM level, and then use a virtual TPM to log application and event measurement. This has the advantage of speed, as the physical TPM is not used for each measurement. More information about the platform's current state can be recorded, beyond just the virtual machine hash. However, it relies heavily on the trustworthiness of the operating system, and a potential exploit could report an entirely false log, as application hashes are not validated through PCR attestation. It also has all the whitelisting issues associated with VM and application level measurement.

Another hybrid option is to combine all of the above three options and measure at every level, using different PCRs for each. This gives three levels at which attestation is possible, and the challenger can chose which level of information to ask for, a potentially lower impact on privacy. Sealing remains viable against the VM measurement, but detailed behavioural state can be ascertained from the event and application measurements. It also provides *defence in depth* as the OS policy provides one level of guarantee, and the other measurements can corroborate its effectiveness. There are some disadvantages, however. The whitelist grows at a much faster rate, as there are measurements of all components. The usefulness of the event reports will depend on the isolation and security of the operating system, as well as on the challenger's understanding of the individual applications. The interpretation of PCR measurements will be even more complicated, and there may even be inconsistencies between different levels of measurement.

Summary

The implications of this analysis are that the most practical integrity measurement level will depend on the goals of the security system, and the properties of the operating system. If a detailed audit of behaviour is required, then a more fine-grained approach should be taken. This might be the case when attesting a single-purpose platform, such as a web service. If management, flexibility and scale are more important, such as for user workstations, then VM-level may be more suitable. However, if the application-level security policies of the operating system are trusted, then VM measurements may be enough for most situations. Similarly, it is only reasonable to have an event-reporting system if suitable levels of isolation can be provided to each reporting application. This may make it impractical for standard operating systems. A final point of interest is that VM measurement of a single-purpose platform (e.g. an application server that offers only one service) that is unable to run any other programs

could be considered equivalent to application level measurement, and so has little additional benefit.

3.3.3 Measuring applications: Strategies

Through a survey of the literature, and experiments, several strategies for performing measurements of the components listed in Section 3.3.1 are given below.

No measurement

The easiest way to deal with a component of the platform is not to measure or record it at all. This is suitable for any file or application that is never read or executed in any way. There are some scenarios, however, where simply `stat`-ing a file may be enough to make it worth measuring. Some configuration files, for example, can reside in multiple places, and if one exists in a certain location, it may override another. So any attestation of the overridden file will be misleading. A related strategy to *no measurement* is therefore *existence check*, where just a file name and 'yes' or 'no' is recorded. A standard hash would also work.

Binary hash and disclosure

The standard TCG-defined approach to integrity measurement is binary attestation, where each application is measured and then compared to a well-known reference value. However, when a custom application is run, there is no reference value. The only way to attest any meaningful information is to show the content of the file that has been executed (or was compiled to produce the executable). The easiest way of doing this is to make the content available. This does not tell the relying party how they might *use* or analyse the content. It also does not help if the file contains sensitive information, such as a password.

In some cases, therefore, the attesting party will wish to make most of the file public, but keep a portion of it confidential. In such a scenario, a *mask* might be used, as suggested by Munetoh [141]. The confidential parts of the file are first erased (or replaced with blank characters) and then measured and made public. This would work for a system where passwords and usernames are specified in plain text, along with other, less sensitive configuration settings. However, if the confidential aspect of the file is the part which has a property to establish (for example, if the aim is to demonstrate that a system uses strong passwords) then a different techniques must be used. Applying a mask is inherently an application-specific decision, and, as such, this method can only be implemented on a per-application basis. This would break any system-wide scheme.

Binary hash and blind analysis

As an extension to hash and disclose, any custom file (be it a configuration, script, or executable) can be analysed rather than completely revealed. This is discussed further in Chapter 7 to demonstrate properties of Java applications through source code. The advantage is that

disclosure is not necessary; the disadvantage is that a full platform is required to produce any analysis credential. There are also several properties which cannot be established through a static analysis.

Measurement by agent

Rather than having the attesting platform measure files and the relying party analyse their trustworthiness, this job could be delegated to an agent running on the attesting machine. As suggested by Yoshihama et al. [239], the attesting platform demonstrates that the agent is running and then lets the relying party query it:

> 'An example of such an agent is a local daemon that reads system configuration files, and composes a structured message that describes the properties of the configuration (e.g., network setting, minimum password length, etc.).'

This then becomes binary measurement of an executable (the agent) plus either runtime querying or an event-measurement (see item 3.3.3). This is similar to Semantic Remote Attestation [85], but without necessarily being integrated into a single virtual machine. The disadvantage of this method would be that either multiple agents would need to be running, or one would need to understand many different file formats. Furthermore, an additional process (particularly one designed to read system files) is an opportunity for runtime compromise. The TCG Platform Trust Service (see Section 6.6.6) is a good example of a system-wide measurement agent.

Event reporting

Configuration files do not have to be explicitly measured before interpretation. The alternative is to measure the *events* the application produces instead. This has a number of advantages. The configuration file can have any semantics and format without requiring the verifying platform to understand it. Furthermore, the behaviour is the actual property that matters, and is also the aspect that is being measured. Equivalent configuration files will result in the same behaviour and therefore the same attested result. Finally, the application will know best which events are important, and which configuration settings are most relevant and need measuring. This puts the work in the hands of the developer, who has the expertise.

One problem with this idea is that it requires considerable effort on the behalf of the developer, who must modify their application. It could also be argued that this strategy violates the measure-before-load principle of trusted computing. Although no configuration file would actually be executed, a malicious one could potentially cause a buffer overflow (or similar) which would avoid the logging and never be noticed in an attestation. Care must also be taken to make sure that the events cannot easily be forged by another process. Any process with access to the PCR will be able to extend the same entries as this process, which would be misleading. However, this could be avoided by providing some access control at a higher layer, or by limiting event-based reporting to occur only immediately after the application is loaded.

3.3.4 Summary

Several methods for measuring integrity exist, and the best method will probably depend on the component being measured. That means there will be several ways implemented on one platform, which makes the job of the challenging party more difficult. Moreover, different types of platform will benefit from different approaches, so it may be that services require a different method (perhaps higher granularity) to clients. This motivates the idea that there is room for improvement with attestation for services.

3.4 Gap Analysis and Conclusion

From the literature review and analysis in this section, the following gaps were spotted in the current state-of-the-art.

Perhaps most surprisingly, the full impact of the software-update problem has never been investigated. Some statistics on the number of process and components of a platform exist [179, 111, 61], but the impact over time of upgrades requires further assessment. This may cause potential trusted computing adopters to be put off, when the impact is low. This is particularly as most existing analysis concentrates on client machines rather than servers [179, 74, 10]. In the next chapter *an analysis of the impact of software patching on a web service over time* is provided.

While there have been several attempts at reducing the TCB on general platforms, including reducing the OS [7] and JVM [151], little work exists on solving the middleware problem for web services. Indeed, many solutions rely on customised language virtual machines [85, 4], or large components in the trusted computing base [142, 116]. These can provide the necessary functionality, but have all the problem of increasing the runtime attack surface. A clear gap in the literature is *providing trusted functionality without a large TCB overhead*.

Many of the solutions described in Section 3.2 and in the approaches outlined in Section 3.3.2 use Platform Configuration Registers at different levels – in the application, OS and boot layer – but do not go into great detail about how to interpret attestations [147, 202]. Measurement from the Operating System onward appears to be a particular oversight. Although the IMA system exists, and the TCG have defined the Platform Trust Service specification, how to develop applications so that their configurations are easy to attest remains an open problem. This is complicated by concurrency, user login [111], and the multiplexing of PCRs. A significant contribution of Chapter 6 is a *unified approach to attesting applications and configuration*.

Another missing component is the ability to go from software identity to properties. This functionality is assumed by many systems [178, 35] but few go into details about how it can be achieved. Approaches such as hooking system calls [83] are one method, but the relationship between this and behavioural properties is unclear. Existing software assurance mechanisms (as discussed in Chapter 2) might be applicable, but to the best of the author's knowledge have never actually been applied. Therefore, the second opportunity is *connecting program analysis and attestation*.

There are several others issues which need to be solved to make attestation more viable. Better operating systems would certainly help, as would a better infrastructure for refer-

ence integrity measurements. Performance and compatibility are also holding back adoption. However, these problems are beyond the scope of this dissertation, which seeks to assess and improve the *feasibility* of service attestation, rather than the direct implementation of trusted computing systems.

The four gaps identified in the literature are the main motivation for the next four chapters. By providing a solution to these problems, web services may become capable of attesting to their own trustworthiness. In Chapter 8 the problems and gaps discussed in this chapter will be used as reference to evaluate how successful the proposals have been.

Chapter 4

Analysing Web Service Attestation

Several papers on trusted computing make the point that attestation is largely infeasible, in part because the number of potential software configurations that might be reported [61, 178, 85] is too large. However, most literature focuses on *client* machines running dozens of applications. Web services (and servers in general) are not considered, and may be more practical.

In this chapter, the difficulty of attesting a web service is analysed. First, the properties of a highly *attestable* system are discussed, such as one with a small TCB and few patches. This can be used to identify where attestation is most likely to succeed. Is it in client machines, as most suggest, or in servers and services? In Section 4.2 these conclusions are quantified by taking a web service platform and counting how many integrity measurements are required to measure it, and what the system state it reports looks like. The analysis is concluded in Section 4.3. A shorter version of the results and analysis in this chapter was originally published in conference proceedings [122].

4.1 What Makes a System Easy to Attest?

The impact of some of the issues described in the last chapter can be quantified, such as *trusted parties*, *performance* and *whitelisting*, but the others are more difficult to analyse. It is possible, however, to identify the best and worst-case scenarios for each of them. Knowing the scenarios where attestation is least practical can then inform an analysis of the practicality for service attestation, as discussed in the rest of this chapter.

Best and worst-case scenarios are as shown in Tables 4.1-4.7. These give circumstances where the particular attestation problems are most and least significant, either because they are mitigated through the use of certain technology or because there is less of a threat in the scenario. For example, Table 4.1 shows the best-case scenarios for attestation with respect to the privacy issues outlined in Section 3.1.5. If the measurement logs remain confidential through secure transport sessions or restricted networks, the privacy issues are much less important as it is not possible for an third party to learn the platform configuration. Similarly, if the challenger is unknown, then this implies that they may misuse the reported measurement log

or disclose it to a third party, making privacy more of a concern.

Some of the features given in these tables simply depend on good design or implementation details, for example, using robust software, caching, and compatibility. However, there are some underlying principles which will be dependent on how attestation needs to be used. There will be situations where a large amount of software *must* be attested, or where a complex property needs to be established. The next section will analyse which scenarios have more in common with the best-case than the worst-case features.

4.1.1 Where should attestation work best?

Given the properties listed in Tables 4.1-4.7, there are many reasons to consider attestation more or less practical for any given platform. There are some niches that seem particularly promising, such as attestation of single-purpose systems, like online banking virtual machines [206] and games consoles [11]. Indeed, many games consoles, such as the Sony PlayStation 3 support the secure boot process [191]. These work because they are rarely updated and users do not require a large, flexible range of software, minimizing the whitelist problem. However, some platforms in service-oriented architectures are more suitable than others and the rest of this section provides an analysis of them.

Home client platforms

Computers used at home for online banking, games and media playing may benefit from attestation. Banks would like to establish that no malware is running before allowing customs access to their online accounts. Game servers would like to make sure players are not cheating through unfair local program modifications. Media companies are keen to prevent unauthorised sharing of music and video by users, through Digital Rights Management. This ultimately relies on the state of the user's platform. Therefore, there is a good case for attesting a general purpose platform, perhaps to one or more remote servers.

The advantages in this scenario are that the client machine will be rebooted often, so malware will need to load at boot time, making integrity measurement an appropriate mechanism for identifying it. This also reduces the chance of runtime attacks. Home operators are likely to be running standard, pre-compiled software, which makes it easier to identify. It may already be managed in one package-management system, which provides an automatic whitelist. Attestation is likely to be one or two servers, which will not impose a performance penalty to the home platform. It is also possible that the property being attested is relatively simple, such as 'unmodified application in use' although this would be situation-dependent.

Disadvantages are numerous. This is the scenario feared by privacy advocates, as the media provider might mandate certain software configurations which the client is unwilling to use [166]. The freedom to run and compile any application is important to many people. The range of software and hardware available is huge, and compiling a comprehensive software whitelist (with any semantic value) would be difficult. Furthermore, at any one time users may be running (or have run) web browsers, email applications, word processors, games, and

Best-case features	Worst-case features
Measurement log remains confidential through encryption or other means	Uncontrolled measurement disclosure
Attesting to a trusted challenger	Attesting to many unknown challengers
Attesting robust software	Attesting software with known exploits
Attesting few components	Attesting many components

Table 4.1: Privacy: Best and worst-case scenarios for attestation

Best-case features	Worst-case features
Attesting few components	Attesting many components
Attesting software with a known, proven property	Attesting under-specified applications
Open source software, allowing white-box testing	Only black-box testing
Relying party requires only one property, dependent on one component	Trustworthiness dependent on all components
Behaviour only depends on executables, no user input or config files	Configuration files, data and runtime events can affect platform behaviour
Property easy to express and demonstrate conformance	Non-functional or high-level property
Platform has known trustworthy configurations	Unknown whether platform configurations are trustworthy
The combination of components on the platform is unimportant	The order and combination of applications is important

Table 4.2: Semantic Gap: Best and worst-case scenarios for attestation

Best-case features	Worst-case features
Attesting few, small components	Attesting numerous large components
Platform is offline, or is only connected to an internal network, or has very few inputs	Platform has many inputs and open ports
Attesting robust software	Attesting software with known exploits
Platform has been hardened specifically against runtime attack	Attesting off-the-shelf software with no security consideration
Platform inputs are validated and have simple data structures to parse	Platform has many interfaces, each requiring a large amount of software to handle
Platform restarts regularly	Platform stays running indefinitely
Platform upgraded when new exploits discovered	Platform cannot apply patches quickly

Table 4.3: Runtime: Best and worst-case scenarios for attestation

Best-case features	Worst-case features
Attesting few components	Attesting many components
Configuration of attesting platform already known to challenger	Challenger must assess previously unknown platforms
Components are rarely changed or patched	Components change regularly
Changes to components are published, along with expected hash values and properties	Components changes (and reasons for them) are not made public
All components are part of one package management system	Applications are downloaded and installed individually, from many sources
Executables have a common hash value	Executables are often recompiled, patched or customised, making their correct hash unknown to challengers
One whitelist only, handled by a dedicated party	Many different relying parties, each with their own whitelist to update
The particular combination of components on the platform is unimportant	The order and combination of applications matters

Table 4.4: Whitelisting: Best and worst-case scenarios for attestation

Best-case features	Worst-case features
Shallow certificate hierarchy, many issued from same authority	Large hierarchy, different authorities, multiple vendors
Reference measurements signed by one authority	Many vendors and authorities produce reference measurements
Few applications and few platforms	Attesting multiple machines with different configurations from different vendors

Table 4.5: Trusted Parties: Best and worst-case scenarios for attestation

Best-case features	Worst-case features
Occasional or one-off attestation	Regular attestations by multiple parties
Infrequent use of sealing	All incoming data sealed with PCR-bound key
Opportunities for caching or performing software cryptography	Only TPM-based cryptography possible

Table 4.6: Performance: Best and worst-case scenarios for attestation

more. Many of these have plug-in architectures, and all would need modifications to support attestation. This is an enormous stack of software, running on a commodity operating system, the most popular of which are closed source, allowing only black-box testing. The large number of programs also means more frequent updates, increasing the whitelisting problem further. The number of targets for a runtime attack is huge, and the operating system may offer little memory isolation for each application. This effectively makes the TCB of the system include all applications, as it is a user-level application that would need attesting. The platform will be connected to multiple servers and have many interactions using different protocols, increasing the chance of runtime compromise. In addition, the platform has constant human interaction, which may affect platform state and require reporting. The certificate hierarchy will be large, as there will be multiple vendors, many authorities and third parties. Most importantly, until the operating system provides stronger isolation, the TCB of a client machine will be too big to attest.

All these issues add up to mean that home users are unlikely to benefit from attestation without serious changes to how operating systems and other software are structured. Future attempts to improve application isolation would help, but will not mitigate many of these issues.

Corporate client platforms

Companies often allow remote working from laptops or mobile platforms. Securing these machines is important in order to maintain control over company data and prevent information leaks and viruses. This is the scenario proposed by Sailer et al. [179].

The advantages in this situation are similar to those of the home user. In addition, the number of allowed applications may be far smaller, and the platforms may have been pre-installed to a trustworthy configuration by the company. This will make creating a whitelist and policy much easier. Having a smaller range of possible configurations also removes privacy concerns and increases the trustworthiness of the platform in the face of runtime attacks. There may be a much smaller certificate hierarchy, too, as the corporation may run its own certificate authority and certify its own hardware.

Many of the same disadvantages remain. There is still a large amount of software, with all the problems highlighted in the previous section. Over time, many new configurations will appear, and within one organisation, there may be a need for many applications to be supported. Commodity operating systems will still be used, as will standard web browsers and email clients. These are complex applications with a history of published vulnerabilities.

Overall, the corporate platform is much more amenable to attestation than general home platforms, but still suffers from the use of large, complex software, resulting in a big trusted computing base.

A public web service

One of the goals of web services (and other online interoperable platforms) is to allow dynamic collaborations of multiple services that may reside in any public location on the Internet. For this to be achieved, some services will need to offer guarantees of trustworthy operation, for example, e-commerce systems, cloud services and online banking. Attestation could be used to provide part of this guarantee, demonstrating that no outsider has attacked the system. It could also prove that the server will behave in the manner it claims to.

The advantages begin with a smaller trusted computing base. Compared to a client machine, there are far fewer components, often serving only one purpose. It might consist of a small operating system, middleware, applications and remote data storage. The smaller TCB has many knock-on effects. There will be fewer applications running, so fewer targets for runtime compromise. Whitelists can be smaller, and as only one platform needs attesting, it does not need to cover multiple hardware configurations either. This in turn reduces the rate of updates and patching. In addition, servers seldom need graphical interfaces, or to interact with local users at any time. This means that load-time properties are strongly related to runtime properties. Other advantages include the popularity of open source server software, and the fact that many software components will be hardened by administrators with real expertise, in comparison to a home user. This allows them to run cut-down applications, or even a microkernel operating system. It is also possible to separate the service provider from the hardware, through cloud computing or remote hosting providers. This eliminates the possibility of hardware-based insider attack. Finally, servers are likely to be providing (or guarding) important, useful functionality. This makes them much more relevant to attest, as the required effort in attestation matches the reward. Many client machines can benefit from one trustworthy server.

There are disadvantages. More effort is required of the end user. They must be able to cope with validating the certificate hierarchy. On the other hand, this is already true of transport security on the web today. They must also have trustworthy software for verifying the attestation certificates. More importantly, servers have availability requirements, and are therefore rebooted infrequently. This makes attestation less likely to spot malicious activity. However, in the world of web services, servers are often designed to be *stateless*. This means that services could be restarted more frequently, as there would be no internal state to save and reload. Other disadvantages include the untrustworthy nature of the challenger, who might use platform configuration information to identify targets for attack. Performance is reduced, as servers may need to attest to many clients. Furthermore, servers use many configuration files and scripts which are harder to attest properties of. This is also true of client machines, however. The last issue is that server middleware is often large, with a huge code base, making it a big target for runtime attacks. In web services, this middleware must also interpret many complex data formats, such as SOAP, WSDL and SQL.

Internal corporate web services

The advantages and disadvantages identified with external services remain largely true of internal service-oriented architectures, although the risk/reward ratio changes slightly. Some issues are mitigated – only trusted clients will attest the server, the certificate hierarchy and client software can be managed by the internal IT department, and whitelists are also easier to manage. However, the benefit is smaller, as an internal service should be largely isolated from the outside, where many intruders originate. The threat from a malicious insider can still be reduced, but may be less significant, as many other techniques exist for mitigating this.

Summary

From this high-level analysis, it appears that all attestation scenarios are difficult, but that arguably public servers offer the best trade-off between trustworthiness added and practical problems to overcome. It also seems that the severity of many of the problems depends largely on how many applications are in the platform's TCB. In order to test the hypothesis that a web service is a good match for attestation, the following section will look at one of the key metrics: how many pieces of software it must attest.

4.2 Quantifying the Software Update Problem

An experiment was designed to quantify the difficulty of attesting a typical web service platform 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 the experiment had to take into account the *rate of change* of the platform. Information from two sources was used: the Ubuntu Linux package repository [224] and the Sun website. A two and a half year period (June 2006 to January 2009, inclusive) was studied. The overall plan was to install a 2006 software stack, modify it to support authenticated boot and then count the size of the measurement log it produced. After this initial baseline count, the platform would be updated in line with released software updates, counting the number of new integrity measurements (hashes) required after each update. The rest of this section details how the web service platform was configured, and the methodology for counting updates.

4.2.1 Methodology

The experimental platform was based on popular web service software from 2006 including the Ubuntu Linux 6.06 operating system, OpenJDK Java runtime and the Glassfish application server. The service was written in Java and had just one function: attesting to clients using standard SOAP requests and responses. No further functionality was considered in order to make this experiment reasonable for all generic web services.

The BIOS and bootloader were measured but the number of measurements was not included in the results. This is because new Intel and AMD processors support a late launch feature (see Section 2.3.5), which combined with a bootloader such as OSLO [103] makes verifying these components unnecessary. Furthermore, they add only a constant, relatively small number to the final results.

Several modifications were made to standard software in order to support authenticated boot. These were based on the most popular and reliable trusted computing libraries available. A custom version of the Ubuntu 6.06 kernel was compiled, initially version 2.6.22.1, complete with the IMA [180] patch to measure executables and kernel modules. A modified version of the OpenJDK based on work by Dietrich et al. [58] was then installed on it, along with standard versions of the Glassfish Application Server, to run a simple web service which answered attestation challenges. The IAIK JTSS [101] libraries were used to communicate with the TPM from the web service. Using this software, attestation requests from another platform we made and the results were recorded in a database. This set of software was chosen for its popularity for servers – Ubuntu Linux is increasingly being used for servers [57] and the Glassfish application server was downloaded 3.5 million times as of June 2007 [161]. This makes the test platform a reasonable case study.

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. The 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. Each application was then analysed to see how it would change after an update. The final output was a timeline, containing all files that were changed and the date the new versions were released. A pro-active administrator is assumed: someone who applies all patches as soon as they are available, but does not upgrade the entire OS distribution.

Operating system. Every version of the kernel that was released in the Ubuntu repositories for the 6.06 distribution was counted. Every new version had entirely new hash values for each kernel module. Because the IMA patch measures every kernel module, the total number of measurements recorded of the kernel was the initial number of kernel modules loaded multiplied by the number of kernel updates.

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 the baseline attestation.

Java. Java updates were handled manually through the Sun website. It was assumed that a new version would be installed whenever available, and that the migration from Java 5 to Java 6 would happen at the first opportunity. Because a customised version of the JRE was being used (compiled from source) to support integrity measurement, it was impossible to install each new version and re-run the attestation process. Instead, a list of files that our custom JRE used was created, and then it was worked backwards to see

which updates modified files on this list. It is anticipated that a few libraries will have been missed in this process, but that number should be small. Versions 5.6 to 5.10 and 6.0 to 6.11 of the JRE were counted.

Glassfish. The libraries and executables associated with Glassfish were counted by installing and running it on the customised JRE. A simple web service was run on each version. This service had only one function: returning attestations when challenged. It was then updated with every core release of a new version of Glassfish, as detailed on the download page of the website, excluding version three, which was still in beta. Because Glassfish was not modified in any way, it is likely that a few libraries were loaded using an unmodified classloader and therefore not included. Again, however, it is expected that this number is small.

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, it was impossible to 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. It was therefore decided to get approximate figures by measuring the total number used, and then estimating an upper bound on how many would be relevant, ignoring change over time.

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

4.2.2 Results

The baseline system had 277 components which were recorded in the integrity measurement log, 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 recorded on the measurement log. Apart from those already mentioned, there were 13 base packages updated, including gzip, udev and e2fslibs.

From the record of configuration files, it appears that a total of 113 were read, with 49 that were either empty or considered unimportant, leaving 64 that might need to be measured. These contained a total of 6370 lines, with just under 5000 in the 64 important files. The vast

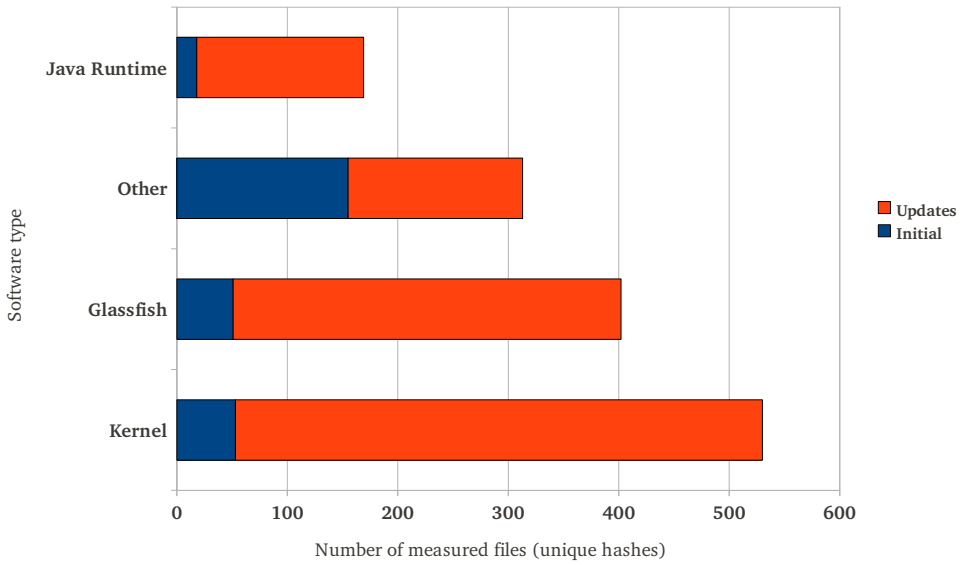

Figure 4.1: Measurements and updates by component

majority of these (3414) were in Glassfish XML documents. It is likely that some of the Glassfish schema files considered significant would never change and could be attested using a hash, making line count unimportant.

4.2.3 Analysis and implications

On assessing platform trustworthiness

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.

For the purpose of identifying running applications and checking their integrity, these results look promising. Any database can store 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, the vendor and patch-level are easy to check, which can be useful when assessing other properties of the platform. For example, Munetoh et al. [142] use this information 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 root kit would be trusted by a remote user. Again, it is assumed that all applications support integrity measurement. These results reinforce the idea that attestation

Best-case features	Worst-case features
Few pieces of software on platform	Large stack of software to attest
Applications do not load executable content	Applications use plug-ins and macros
Applications designed so that loading happens only in one place, making measurement easy to implement	Poorly written applications load files and input data at different places in source code
Applications designed with specific properties in mind, making attestation and verification easier	Applications lack specific security properties
Operating-system level modifications can record loaded content with no application modification needed	Applications load data files, only some of which are executed, making it difficult to manage by the operating system

Table 4.7: Compatibility: Best and worst-case scenarios for attestation

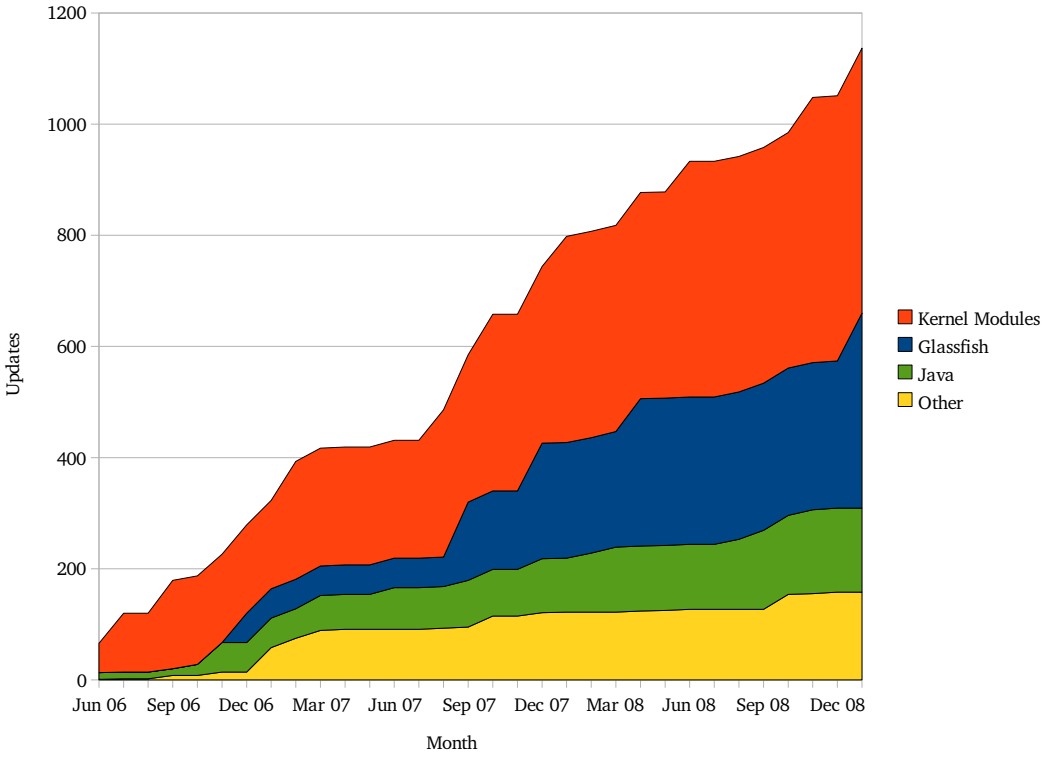

Figure 4.2: Cumulative updates by component over time

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

Table 4.8: Updates applied by month

is suitable for establishing that a platform *did not*, at boot-time, have a root kit installed.

Another property that *can* be attested by the platform is that nobody has logged into a terminal, either locally or remotely. This is because certain executables are run at login, 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 into 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*.

Attestation cannot be used to establish a platform's *correct* behaviour, as none of the hardware or software has been analysed for this. But will the software behave as expected? It might be assumed that this is the case, as Linux, the JVM and Glassfish do generally work in their expected way. However, runtime attacks remain a problem. 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 regularly, and this makes it impossible for attestation to support any claim of trustworthiness. This is a well known criticism of common operating systems [117]. It is therefore not possible to establish trustworthiness because the security state of the server cannot be assess through TCG attestation alone.

In order to move from *identified* to *trustworthy*, the chance of runtime exploit needs to be reduced. This means limiting the number and size of applications running on the server, and improving the quality of the code. In Chapter 5 an approach for doing this is discussed.

On measuring configuration files

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. These results show that a significant amount of configuration must be dealt with, but no existing solutions exist for doing so. Chapter 6 discusses this problem further and presents a solution.

On obtaining reference values

Collecting 1414 measurements would not be difficult for an end user system, but keeping track of which of these entries is trustworthy or requires updating is more so. It is unlikely that every user will want (or be able) to compile this list themselves. The TCG suggest [207] a more sophisticated architecture which has many sources of reference measurements, aggregated into a Reference Manifest Database (RMDB). This is then used by a verifier, who reads each attestation and makes a decision about trustworthiness based on a policy database (perhaps informed by a configuration management tool). The decision is then passed on to the relying party. The TCG infrastructure puts a low verification overhead on each user, but requires several intermediate steps and parties.

These results show that an integrity database could be small, and therefore some of these steps could be combined to allow decisions to be made on the users' own platforms. For this to happen, a complete copy of the integrity database and sufficient quality information about each item in it must be made available. If instead, users downloaded a signed copy of this information from an RMDB and verifier at regular intervals, then these two platforms no longer need to be constantly available, avoiding a potential denial of service attack. The above results can arguably justify this alternative approach. There were updates on 56 different

days, excluding those released on the same day, averaging 17.3 days between each update, with a range of 1 to 68 days. The number of new files per update was often small, between 1 and 88, averaging 20. Being pessimistic, and assuming every update invalidated as many measurements as it validated, there would be potentially 20 measurements every 17 days, with 40 new trust values. This seems a manageable quantity and such an overhead would be reasonable to impose on client machines and central repositories.

Software layers

The rate of updates depends largely on three components: the operating system, language runtime, and service middleware. It is not unreasonable to assume that the service itself will change frequently, too, as the developers add features or fix bugs. This means that the rate of change of the system does not change throughout different logical 'layers' of the system. The operating system, middleware and application are all significant sources of update and change. This might not be the case with a hypervisor layer, but there is no evidence to suggest otherwise. This contradicts assertions made by Marchesini et al. [126] that it is worth splitting software into long, medium and short-lived categories. From this experience, very few pieces of software are long-lived.

Is the right information being analysed?

In one way, these results can be considered an *upper bound* on the number of measurements over this period. This is because an extremely proactive update cycle was assumed which is unlikely to be followed by many administrators with concerns over availability. It would make more sense to limit updates to those with security implications. However, this is difficult to do, particularly in the Linux kernel, where security bugs are not always marked.

On the other hand, these results do only consider measurement, not assessment. As a result, when one element changes, the entire platform (in theory) needs reassessing, as any single executable could invalidate a security property. This means that the assessor's job is not to test every item individually, but to test the whole platform after every update. This will make the testing process much more time-consuming. If a test is written for every piece of software (around 277 at any given time) then they must be rerun and altered for every update batch. As 56 batches occurred, this means a theoretical 15512 test runs, excluding integration testing of the entire system. Of course, this is not realistic (many items will not need testing) and it is likely that only components such as the kernel, modules, JVM and Glassfish would need regular testing. This means a baseline of about 123 applications, and 6888 items of test data, plus integration tests. However the software is tested, the number of components is probably too large for a high level of assurance.

4.2.4 Comparison with client platforms

The comparative difficulty of measuring a standard client platform was explored with a brief experiment using Ubuntu Linux 9.10. The integrity logs were checked twice – once before user

Platform Description	Measurements
Ubuntu 9.10, Pre-login	1859
Ubuntu 9.10, Pre-login, list filtered	1442
Ubuntu 9.10, Post-login	2304
Ubuntu 9.10, Post-login, list filtered	1802

Table 4.9: Client platform integrity measurement count

login and once after user login using the X windowing system. Table 4.9 shows the results. It should be noted that newer versions of IMA and Linux were used, which increased the number of results. To make the logs comparable, however, the integrity measurements were filtered to remove the extra file types reported by the newer version of IMA. Filters removed the `.config`, `.rules` and python scripts, erring on the side of removing measurements where possible. Both pre-login and post-login are relevant, as while a web service may never need to be logged into, a client machine certainly will. The results given do not include any specific applications (such as a web client, email client, etc.) and it seems likely that using these would greatly increase the number of measurements.

Although this experiment was not carried out over time, it does show how much more difficult the problem of integrity measurement is for client machines. A server baseline of 277 measurements means a whitelist only 15% as large as for a logged-in client platform. The implication is that standard TCG approaches are much more suitable for servers than clients.

4.3 Conclusion

From the experiment presented in the chapter, it is possible to conclude that *attestation* of an individual service is entirely feasible. However, there are challenges with making use of the attested information. The testing (or any assurance method) effort is enormous, particularly due to the rate of updates. Furthermore, it is unclear how to measure configuration, or even how to interpret these measurements to establish platform state. There would be little point in performing any software assurance technique on the service code itself (or any individual component) as the amount of code it represents is dwarfed by the large code base of the operating system and middleware. The first step in making attestation more useful is to reduce the TCB of the platform, as this would reduce the number of updates and measurements and make assessment easier.

Chapter 5

Reducing The TCB of an XML Web Service

The size of a system's trusted computing base has been identified as a key metric for assessing suitability for attestation. Unfortunately, an individual web service contains a considerable amount of software, most of it in service middleware and the operating system. However, the measurements taken in Section 4.2 were made of a standard web service and no special effort was made to shrink the example platform. By modifying the configuration, several applications could be removed. This would have a cumulative effect over time, as fewer components mean fewer total updates.

One way in which this could be achieved would be to reduce integrity measurements made by the operating system. Kernel modules could be compiled statically into the kernel, rather than loading at runtime causing an extra measurement. This would result in 53 fewer measurements per boot and therefore 53 fewer new measurements every time a new kernel is installed. Alternatively, when updating the kernel, greater care could be taken to make sure that only critical modules were changed. Both of these methods may simplify or reduce measurements, but ultimately, trusting a large operating system is a fundamental problem [117]. A better approach would be to use a smaller system, perhaps with a microkernel architecture, making it feasible to verify formally [106].

Glassfish libraries were the second largest source of measurements, and alternatives could have a smaller footprint. Unfortunately, it is not an unusually large service environment. Apache Axis2/Java 1.4, which offers similar features, uses only 8 fewer JAR files than Glassfish 2.1. This implies that the amount of functionality is the problem, not the specific implementation.

The size of other core applications could be reduced. The IAIK Privacy CA project [151] uses a system trace to remove all unnecessary class files from their JVM, and a similar method could be used here. This would reduce the application of unnecessary patches. However, there is a limit to the amount of code which can be removed without reducing functionality. This also encourages the hand-crafting of code, which may make it difficult for remote parties

to find comparable hash values. The complexity of data formats such as SOAP and XML mean that a web service platform *must* contain certain large applications.

In this chapter a method for reducing the middleware problem [50] is discussed, with a focus on maintaining compatibility with web service standards. This is the key component of a service that allows for interoperability and this should not be compromised. A solution to this problem is proposed in Section 5.1. Section 5.2 then deals with additional problems, Section 5.3 provides a security analysis, Section 5.4 explains some of the design choices made and Section 5.5 considers performance. Finally, a comparison with similar work is presented in Section 5.6 and the overall benefit of this approach in summarised in the conclusion. A shorter version of this work was originally published in conference proceedings [123].

To evaluate the success of this approach, Section 8.2 presents an example service that has used the proposed solution.

5.1 A Split Service Architecture

Service middleware is complicated in part because it must support complex data formats, as well as other features such as load balancing and auditing. To test this theory, some of the experiments performed in Chapter 4 were repeated using Java RMI rather than SOAP-based interfaces. Exactly the same system was used as in the earlier set-up, but the attesting service was configured to be accessed through an RMI interface. There was a significant improvement in the number of integrity measurements, as the new system required 78 (28%) fewer entries in the 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 makes the idea of removing service middleware attractive. However, doing so would come with a cost: reducing functionality and interoperability. 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 security.

Fortunately, there is a way to use minimal software and heavyweight protocols and features. This chapter explores splitting a web service into two distinct components, one trusted and one not. The untrusted component is at the front-end and can parse the SOAP and XML requests. It can also perform any management features, load balancing, and other complex functions. Messages arrive at the front-end and are forwarded on in a simpler format, such as Java RMI, to the trusted back-end. The back-end provides all the real functionality and logic. In a data processing scenario, the back-end platform could either *be* the data store, or be responsible for contacting it and forming queries. Figure 5.1 illustrates this system, with VM2 as the trusted back-end, and VM1 as the front-end. VM1 receives SOAP requests from the client, and then translates them into RMI for the back-end to process. The client only needs to attest to the back-end, as it has the functionality of interest.

The advantage of this architecture is that the small back-end component is much more reasonable to attest. Having removed all of the software that should not affect algorithmic behaviour, only the programs that *will* are measure and reported. The operating system and

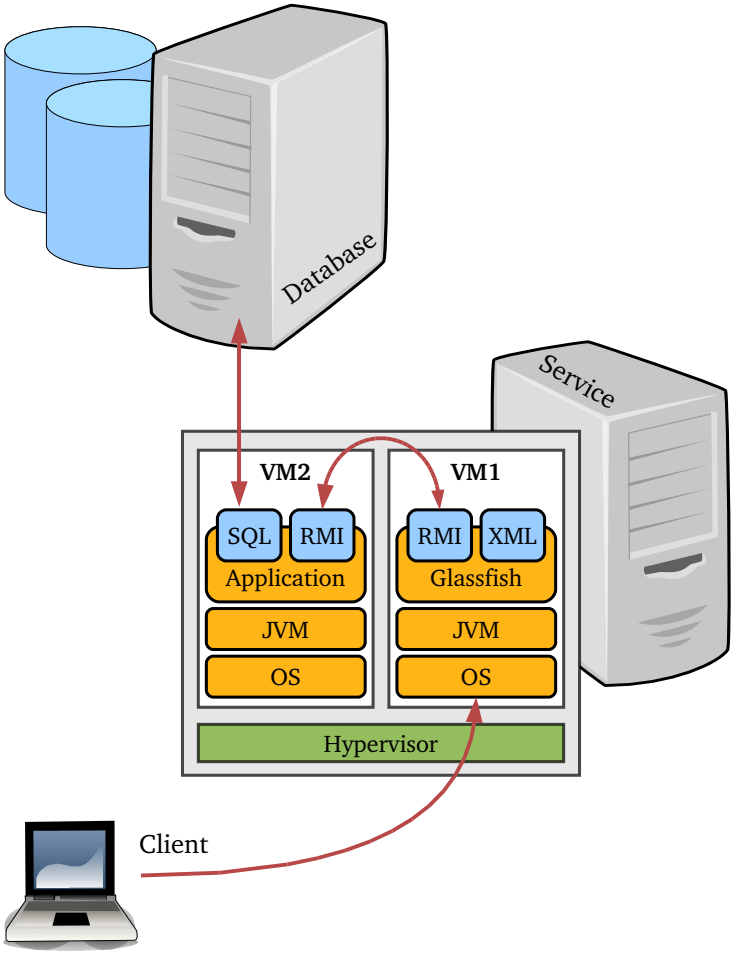

Figure 5.1: The split web service architecture

software can be minimised extensively, as only the exact features required by the platform will be needed. It might even be possible to run a web service application natively on a bytecode processor [132, 240], taking the operating system out entirely. In effect, the back-end becomes a platform designed specifically with attestation-based assurance in mind.

5.2 Implementation Issues

5.2.1 Establishing a secure channel

Having attested the back-end service, a secure channel must be established to guarantee that the service user will be communicating with it. This is difficult, as the untrusted 'front-end' could potentially forward messages on to any host after a valid attestation. This style of *platform in the middle attack* [12] is difficult to avoid and requires the user to know that the platform that attested is the same one that requests are being sent to. This problem has been

discussed many times before (see Section 3.1.8 and [72, 74]). In this scenario, transport-level encryption is inappropriate, as it would prevent the front-end platform from translating and forwarding requests to the back-end. Instead, message-level cryptography as specified in the XML encryption and security token standards [153] must be used. One approach is for the trusted platform to publish a public key, along with proof that the private half is held in its TPM. Such proof can be gained from the TPM CertifyKey command, which uses an AIK to sign the certificate [209]. If the same AIK were used for the attestation process, this establishes that the key belongs to the attested platform.

An initial request for a service's public key could be performed earlier, using the WS-Trust specification [154]. The two-step protocol below shows the user (U), credential repository (C), service (S), service public keys ($PK(S^1)$, $PK(S^2)$) and service AIK ($AIK\text{-}SK(S)_1$). Line 5.1 is a request for a service's public, bound TPM key, and line 5.2 is the response, containing a service key and TPM credential, signed by service's AIK. These steps must be performed in a transport session with a known, trustworthy credential repository:

$$U \rightarrow C: \quad \text{RequestSecurityToken}, S \tag{5.1}$$

$$C \rightarrow U: \quad PK(S^1)\,,\ AIK\text{-}PK(S)_1\,,\ CertifyInfo_{AIK-SK(S)_1}\{|\ PK(S^1)\ |\},\ PK(S^2)\,,\ CertifyInfo_{AIK-SK(S)_1}\{|\ PK(S^2)\ |\} \tag{5.2}$$

Having a known public key for the endpoint means that service requesters can use it to encrypt messages. These can then be forwarded to any platform by the untrusted component, without fear of compromise. Furthermore, any reply message generated by the endpoint can be signed, proving the source of the reply. Of course, it would be necessary to establish a session key rather than relying on just one public key. We therefore propose the following protocol, with the service front- and back- ends denoted as F and S respectively:

$$U \rightarrow F: \quad \text{RequestSecurityToken}, AIK\text{-}PK(S)_1, nonce_U \ \text{(SOAP)} \tag{5.3}$$

$$F \rightarrow S: \quad AIK\text{-}PK(S)_1, nonce_U \ \text{(RMI)} \tag{5.4}$$

$$S \rightarrow F: \quad Quote_{AIK\text{-}PK(S)_1}\{|\ pcr_{0-15}, nonce_U\ |\} \ \text{(RMI)} \tag{5.5}$$

$$F \rightarrow U: \quad Quote_{AIK\text{-}PK(S)_1}\{|\ pcr_{0-15}, nonce_U\ |\} \ \text{(SOAP)} \tag{5.6}$$

$$U \rightarrow F: \quad \text{Method}(\,SYMENC_K\{|\ arg_1, arg_2 ...\ |\}\,), ENC_{PK(S^1)}\{|\ K\ |\} \ \text{(SOAP)} \tag{5.7}$$

$$F \rightarrow S: \quad \text{Method}(\,SYMENC_K\{|\ arg_1, arg_2 ...\ |\}\,), ENC_{PK(S^1)}\{|\ K\ |\} \ \text{(RMI)} \tag{5.8}$$

$$S \rightarrow F: \quad \text{Reply}, HMAC(SK(S^2)\,,\ \text{reply}\,) \ \text{(RMI)} \tag{5.9}$$

$$F \rightarrow U: \quad \text{Reply}, HMAC(SK(S^2)\,,\ \text{reply}\,) \ \text{(SOAP)} \tag{5.10}$$

Line 5.3 is the WS-Attestation request [239] to the service with an already-known AIK and nonce. Lines 5.4 is an attestation challenge, and lines 5.5 and 5.6 are TPM Quote responses forwarded to the user, via the front-end. Line 5.7 is the SOAP method invocation with session

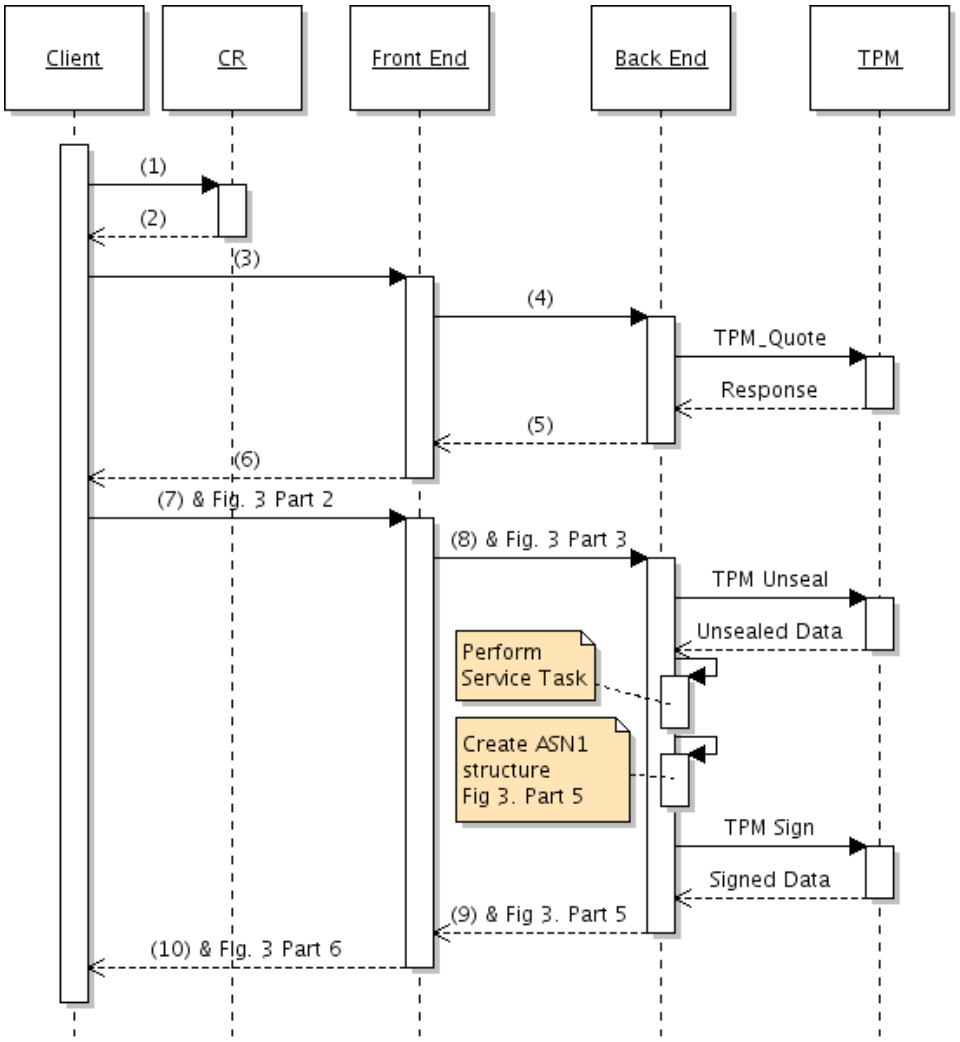

Figure 5.2: Sequence diagram showing steps from the protocols in from Section 5.2.1 and the message formats from Figure 5.3

key K applied to all fields, which is then translated and forwarded via RMI in line 5.8. The reply is generated in line 5.9 and translated again to conform to WS standards in line 5.10.

One additional consideration is mitigating the *platform reset attack*, where the platform is booted into an acceptable configuration for attestation, and then rebooted into a malicious one when it receives the actual data. One way of avoiding this is to use a 'sealed-key' [116] approach. This means using a key bound to PCR values in the TPM, and adding PCR details to the key certificate as proof for the remote user. This would allow lines 5.3 to 5.6 of the above protocol to be removed, as run-time attestation is no longer necessary. Alternatively, the monotonic counters could be used to record the number of times the platform has been booted, as suggested by Sailer et al. [180].

5.2.2 Preserving integrity and confidentiality

The protocol described in Section 5.2.1 is simplified in terms of signatures and encryption. Decryption of incoming messages, and signing of the result, must be performed on the back-end platform, as only it has access to the TPM-stored keys. However, this means that only individual fields can be encrypted, not complex XML structures, as the back-end has no way of processing the XML. This is an important limitation. An attacker now has the opportunity to re-order fields, as nothing binds the content of the field to its location in the document. If the encryption is just of the field itself, then it will also be vulnerable to replay, as no freshness information is present.

Similarly, the response from the back-end service should be signed, but as the front-end must translate to XML, the signature cannot be of the entire response. One alternative would be to sign a digest of the important individual fields. However, this has the same issues as with encryption.

To provide both freshness and structure to the elements, without breaking web service standards, fields must be added to the internal methods and the response. The requester must know that the endpoint was given the correct input, and that any result has not been modified or replaced in transit. This implies that the response should contain a hash of the original input, result and a nonce. To avoid the endpoint from needing to process XML, a set of identifiers can be included internally, linking the expected XML structure to the internal fields. The identifier-result structure is then signed by the endpoint, and included in the response. The example in Figure 5.3 demonstrates this system. Note that the response structure and labels are hard-coded, and not calculated from the incoming message. The verifying party can then compare the request and result against the arguments and result the endpoint declares that it has used.

XPATHS have been used as identifiers as these should be predictable and easy for the verifier to process. The identifier-result structure can be described using a syntax close to ASN.1. The combination of XPATH and ASN.1 allow the description of an XML document without the platform needing to be able to interpret or process it. In many situations this will be more complex than necessary — for example, if only one field is encrypted originally, or if the result is a single item.

The property established by this system is that *if* the final response contains a signed structure that correctly described the user's input, *then* no man-in-the-middle could have re-ordered fields and the service will have responded in the expected way. This does not mean that all requests can be trusted in advance – to do that, an additional phase is required. The message must be sent to the service, a validating reply message received, and then an encrypted 'commit' message must be sent to the service, to confirm the result.

The protocol in Section 5.2.1 also does not provide any confidentiality of the result of the web service request, only integrity. In order to protect message confidentiality, it would potentially (depending on key sharing assumptions) sufficient for the back-end to re-encrypt the result using key K. It is also assumed that any data sent to the back-end platform will be securely deleted after use.

1) Original SOAP Request

```
<soap:Envelope ... >
  ...
  <soap:body ... >
    <m:Entry>
      <m:from>
        Joe Bloggs
      </m:from>
      <m:content>
        Joe Bloggs is a patient
        at Area Hospital...
      </m:content>
      <m:nonce>
        36829463846238
      </m:nonce>
    </m:Entry>
  </soap:body>
</soap:envelope>
```

2) Encrypted SOAP Request

```
<soap:Header>
  <wsse:Security>
    <xenc:EncryptedKey>
      ...
      <ds:KeyInfo ... >
        <ds:KeyName>
          Endpoint Pub Key XYZ
        </ds:KeyName>
      </ds:KeyInfo>
      <CipherData>
        <CipherValue>
          [Encrypted Symmetric Key]
        </CipherValue>
      </CipherData>
      <ReferenceList>
        <DataReference URI='#content'/>
        <DataReference URI='#name'/>
      </ReferenceList>
      <CarriedKeyName>
        EndpointKey
      </CarriedKeyName>
      ...
    </xenc:EncryptedKey>
        ...
  </wsse:Security>
</soap:Header>

<soap:Body>
  <m:Entry>
    <m:from>
      <xenc:EncryptedData Id="name">
        <xenc:CipherData>
          <xenc:CipherValue>
            [Encrypted Name]
          </xenc:CipherValue>
        </xenc:CipherData>
      </xenc:EncryptedData>
    </m:from>
    <m:content>
      <xenc:EncryptedData Id="content">
        <xenc:CipherData>
          <xenc:CipherValue>
            [Encrypted Content]
          </xenc:CipherValue>
        </xenc:CipherData>
      </xenc:EncryptedData>
    </m:content>
    <m:nonce>36829463846238</m:nonce>
  </m:Entry>
</soap:Body>
```

3) RMI Request

```
MessageResponse response =
 endpoint.submitEntry(
  [encryptedSymmetricKey], // session key
  "Endpoint Pub Key XYZ",  // Endpoint TPM key ID
  [Encrypted Name],        // encrypted field
  [Encrypted Content],     // encrypted field
  36829463846238           // nonce
 );
```

4) ASN.1 style response structure

```
messageInfo MessageInfo ::= {
  input {
    encrypted-symm-key  [encryptedSymmetricKey],
    pub-key-id            Endpoint Pub Key XYZ  ,
    variables  {
      { field-xpath     //m:Entry/m:from  ,
        field-value   [Encrypted Name] },
      { field-xpath     //m:Entry/m:content  ,
        field-value   [Encrypted Content] },
      { field-xpath     //m:Entry/m:nonce  ,
        field-value   36829463846238 }
            },
  result {
      { field-xpath  //m:EntryResponse/m:Success,
        field-value  1 }
      }
}
```

5) RMI Response

```
return new MessageResponse (
  result,
  messageInfo,
  SHA1( messageInfo ),
  Sign( SHA1( messageInfo ) )
  // signed with endpoint private key
);
```

6) SOAP Response

```
<soap:Envelope >
  <soap:Header>
    ...
    <Signature ... >
      <ds:Signature ... >
        <ds:SignedInfo>
          ...
          <ds:Reference URI="#MsgVerification">
            ...
            <ds:DigestValue>
              [SHA1( messageInfo )]
            </ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>
          [Sign( SHA1( messageInfo ) )]
        </ds:SignatureValue>
      </ds:Signature>
    </Signature>
  <!-- Key information included here -->
  </soap:Header>

  <soap:Body ... >
    <m:EntryResponse>
      <m:Success>1</m:Success>
      <m:Verification id="MsgVerification">
        [messageInfo]
      </m:Verification>
    </m:EntryResponse>
  </soap:Body>
</soap:Envelope>
```

Figure 5.3: Service request and response transformations

5.2.3 Changes to services and middleware

For the front- and back-end components to communicate, incoming SOAP messages must be translated by the front-end to the internal protocol. This is part of the functionality provided by JAX-WS [97], turning SOAP into RMI, but in this case the processing occurs on a different platform to the translation. Complications arise when using encryption, however, as any encrypted messages cannot be translated, as the front-end does not have access to the decryption key. Instead, they must be forwarded to the back-end. This means that middleware such as Glassfish [161] must be simplified to pass on encrypted messages and any session keys. Similarly, signed results must be converted by the middleware to conform to SOAP standards, without needing any re-encryption. The example in Figure 5.3 demonstrates this.

5.3 Security Analysis

5.3.1 Web service threats

Demchenko et al. [55] and Bhalla and Kazerooni [18] identify key threats to XML web services. These include misuse and theft of user credentials, snooping on unencrypted SOAP messages, maliciously formed input (XPATH queries, SQL injection) exploiting XML parsers and validators, WSDL enumeration, poor site configuration management and error handling. The proposed system reduces the impact of some of these issues, in comparison to a standard web service endpoint that also uses message-level encryption.

5.3.2 Threats mitigated

Most significantly, threats from XML and SOAP parsers are eliminated in this architecture, as they can only compromise the untrusted front-end. These threats are significant, and several attacks have been published on XML parsers. Microsoft XML Core Services had a buffer overflow exploit allowing remote code execution (Secunia Advisory SA22333), and five SOAP server XML parsers had denial of service issues in 2003 (Secunia Advisory SA10398). Of course, vulnerabilities in the parser used to communicate between front- and back-end components would still have an impact, but the protocol is significantly less complex, and few vulnerabilities in Java RMI (for example) have been published.

Similarly, vulnerabilities in popular web service application servers, such as Glassfish and Apache Axis 2, would be mitigated in this architecture. The attack surface is smaller on the untrusted platform and should contain fewer vulnerabilities.

The use of remote attestation helps users avoid services that have poor site management and misconfiguration, assuming integrity reporting covers these components. This is true of any attestation-enabled platform, but this architecture reduces the number of components to report upon, thus reducing complexity and making it easier for a verifier to establish the properties he or she wants. Furthermore, as the TPM contains the encryption keys, no unencrypted SOAP messages can be copied. If keys are sealed to the precise PCR value of the endpoint (including

all software) then users can be assured that data encrypted with this key will only be processed by the right software and platform.

5.3.3 Remaining issues

Some problems remain. Most importantly, though the front-end service may be untrusted, it can impact the availability of the service, resulting in a denial of service attack – as the service has only been *split* into two components, rather than increasing the amount of software, and is no worse than before these modifications. However, a malicious front-end could forward messages and betray unencrypted secrets.

Other attacks that have not been mitigated include credential theft and error handling. Improper use or storage of credentials may remain a problem, regardless of how the server is structured. Error messages revealing too much information will also not be changed, although this system presents two different opportunities for errors to be censored, at the front- and back-end. Furthermore, these two issues are related to the quality of design and implementation of the end service, a property which will be easier to assess in this architecture, particularly in combination with work discussed in the following chapters.

5.4 Observations and Design Choices

5.4.1 Composite services

Composite web services are common, as one service may be an abstraction for a more complex workflow involving several separate 'sub' services. This presents a problem for this architecture, as the back-end platform is designed to only communicate with the front-end, and not have the capability of communicating in higher-level protocols such as SOAP. A solution (as used in Section 8.2.2) is to allow the front-end to proxy and translate any external communications. However, this has a negative impact on performance, and increases complexity. Furthermore, the key management overheads may become large, as each inter-service communication will need to use a different key. These are largely technical challenges which may be considered a reasonable trade-off against the increased trustworthiness of the overall system.

A more fundamental issue is that an attestation of one service, when in fact several are being used, has significantly lower value. If one of the sub-services is behaving maliciously, the user will not be able to tell. Instead, the chain of trust must extend to all of the component services. Furthermore, the end user must be able to tell which services have been used, how they have attested, and what results they provided. Establishing this through an attestation may be difficult.

This issue is analogous to problems with credential management in composite web services. When passing credentials to a service, should the service re-use these credentials when contacting others, or should it use its own? Similarly, is verifying the integrity of the top-level composite service sufficient, or should all sub-services be checked as well? One solution to this problem in grid systems is delegation [116]. Grid nodes are only selected if they guarantee

to pass on jobs to nodes with platform configuration values present on an agreed whitelist. Such a system might work here, but a more comprehensive solution would be required to demonstrate to the end user that this selection was happening. A trustworthy message router (or a trustworthy endpoint *selector*) as described by Watanabe et al. [234] might be a suitable mechanism.

5.4.2 Multiple back-end instances

One advantage of splitting the service is that one front-end can talk to multiple back-end servers. This would be useful for load balancing. Furthermore, it would allow the back-end systems to be restarted frequently, a useful property for attestation. An additional benefit is that individual service functions can be further isolated into different virtual machines. In this way, one logical service can use different virtual machines. This would be useful if many of the functions were simple and one was much more complex, for example. Similarly, the front-end service can be multiplied for performance or availability reasons.

5.4.3 Comparison to XML firewalls

This design is remarkably different from an XML firewall [115], one of the more common techniques for securing web services. An XML firewall filters incoming SOAP messages for malicious input, based on a number of criteria, such as oversized payloads and SQL injection. The proposed method does not rely on any filtering, as messages are translated to RMI before being forwarded. If a malicious messages was designed to crash the SOAP parser, only the untrusted host would be affected. If it was more sophisticated, and designed to exploit a bug on the trusted host, then there would be a problem. In Chapter 7 application security itself is considered, as are methods for providing automatically generated input validation.

5.5 Impact on Performance

The proposed architecture will have a performance overhead. The precise impact will depend on what it is compared to, and Figure 5.4 shows flow charts for four different systems, with the processing steps that each involves. All services must send and receive messages from the network, parse and format the results, as well as doing the required task. Introducing encryption and signing will add another two steps, and if the keys are held in the TPM, this will involve communicating with it as well. The proposed architecture only adds two additional stages at the server – the RMI communication between platforms.

From existing literature, it is apparent that the use of WS-Security is a significant overhead. Gray demonstrates a factor 100 [80] slowdown when using WS-Security on a single machine. Although this is not directly applicable to this architecture – as the web service does not decrypt the messages directly – it seems reasonable to assume a similar performance hit. Furthermore, Gray shows that RMI invocations are generally an order of magnitude faster than WS-Security enabled XML web services, so the additional RMI step can be expected to have a relatively small

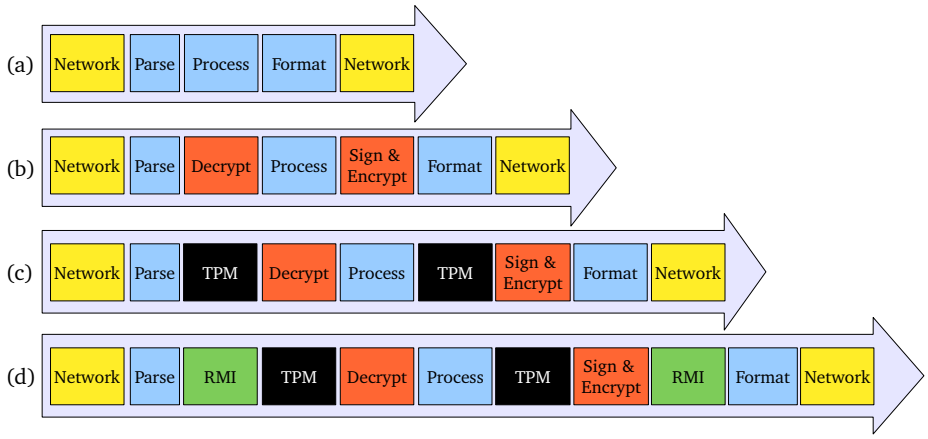

Figure 5.4: Flow chart for four different service architectures, showing (a) no encryption (b) standard WS-Security (c) TPM-enabled cryptography and (d) the proposed TPM-enabled split-architecture.

impact on overall round-trip time and latency. The figures given show complete invocation and message time of just over 1 millisecond for RMI. The split-architecture system involves two such invocations, comparing favourably to the 100s of milliseconds given for WS-Security. And should the front- and back- end services be hosted on the same platform (such as in Figure 5.1) then there is room for more optimism.

The impact of using the TPM is worth considering, although the number of TPM commands is not necessarily *increasing* in comparison to a web service that already uses a TPM for message-level cryptography. For each message, the TPM must decrypt a symmetric key using a key bound to the TPM, and then sign a digest using another bound key. The overhead was simulated to measure how long the TPM took to unseal a 128 bit value using a TPM storage key, and to then sign a 20 byte digest with a TPM signing key. Both keys were stored in the TPM and bound to PCR values. The results were calculated with an Infineon 1.2 TPM and used Brent Boyer's Java benchmarking tool [22] with the IAIK JTSS libraries. Theses operations took an additional 800ms, which would be added to the round trip time of each message. When run individually, both steps took approximately the same amount of time, the bottleneck being communication with the TPM.

There are several ways in which this can be optimised. Firstly, this task is parallelizable, as several back-end platforms can be used, each with its own TPM. This would improve throughput, although clients would need to encrypt the symmetric session keys (K in line 5.7 of the protocol in Section 5.2.1) with several different public keys. Making a different trade-off, the same session key could be used repeatedly for messages sent to the service, which would eliminate subsequent unseal operation on messages from the same client. Furthermore, a key generated from the session key could be used as an alternative to signing with the TPM, meaning only one TPM operation in total. The disadvantage to doing this is that the key is stored in unprotected memory for a significant period of time, making it more vulnerable to compromise. Further optimisation may be possible with virtual TPMs, operating mostly in

software. It is expected that future versions of the TPM will be faster [221], reducing this problem.

5.6 Comparison With Related Work

Minimising the trusted computing base of a platform has been discussed frequently in the literature. Wei et al. [236] split the Apache Axis2 web service middleware into two components, one trusted and one untrusted. They suggest that incoming messages should be intercepted by a 'message splicer' which replaces sensitive information with references. All operations on sensitive data must then be performed by the trusted component, as only it has access to the real data. The untrusted and trusted components are isolated in separate protection domains. This solution presented here is similar, but taken further, allowing a remote user to gain assurance in the web service, rather than just hardening the internal structure. Furthermore, this system considered how messages would be sent between the user and service, noting that a server-side message splicer could not be considered trustworthy by a cautious remote user. The application of trusted computing to this issue seems an essential part of the solution.

Similarly, Jiang et al. [98] attempt to mitigate the threat from malicious insiders by using a secure co-processor, the IBM 4758. This acts as a guardian, performing some important parts of the functionality of the web application. Users can establish a secure session with the guardian and verify they are communicating with it. Their approach does allow for user assurance, but does not work with existing standards for web services. They are also constrained by the use of an expensive secure co-processor, whereas this design can use a standard, low-cost Trusted Platform Module. Furthermore, the threat that Jiang et al.'s system mitigates is that of malicious insiders, whereas the split architecture proposed here will also reduce the risk of external attacks.

Watanabe et al. [234] have an alternative approach, separating the communications component – the 'Secure Message Router' – from the application itself. This SMR is a trusted component, and is used to create high-integrity virtual domains. This is the opposite of the architecture discussed in this chapter, and focuses on establishing guaranteed secure communications, rather than service integrity. It is not clear how service middleware would fit into this scheme. However, having a secure router might be the solution to the problem of composite services. A hybrid approach may be worth exploring in the future.

Cooper and Martin [50] were one of the first to consider the middleware problem and have much the same aims in mind – to reduce the TCB of a grid platform. Their approach is to allow untrusted middleware to pass encrypted data to a virtualized platform which executes it in the presence of a *job security manager*. More details of this approach are given in Section 3.2.9. The key differences are that web service messages are structured, whereas grid jobs are not, resulting in the problems identified in Section 5.2.2. Furthermore, Cooper and Martin must consider the staging of offline data, whereas this system is assumed to be continuously live. Another difference is that this is the attestation of a service with pre-defined functionality, not an arbitrary grid job. The aim in this chapter is to allow for attestation of a service application,

not the infrastructure.

Finally, this approach could be compared to the Flicker [131] and TrustVisor [129] systems described in Section 3.2.14. However, their approach is only applicable to small, security-critical parts of an application, although it does provide a greater degree of minimisation. The technique described in this chapter is more appropriate for providing a slightly lower degree of assurance in a more significant amount of code. A combination of these approaches would be interesting, as the crucial security-critical portions could be further isolated and more accurately attested through TrustVisor with the rest of the system – still important for behavioural guarantees – attested through the method described here.

5.7 Conclusion

The architecture proposed in this chapter demonstrates that the overhead of integrity measurement can be reduced, and therefore attestation may become more feasible as a method for establishing trust in a web service. While only a 30% improvement can be made by reducing middleware, the general approach could be applied to the operating system and Java runtime as well. Unfortunately the operating system is still the biggest problem. There is a great deal of existing literature on the subject of minimising operating systems and improving their trustworthiness [106, 204]. For this reason, as well as due to time constraints, significant modifications to existing operating systems have not been considered in this thesis.

Although the number of integrity measurements has now been significantly reduced, there are more challenges to solve based on the earlier gap analysis. Firstly, it is still not obvious how to interpret attestations to establish system state, even with reduced OS and middleware functionality. Integrity measurement logs only provide a simple, linear overview of the boot process, and say nothing about system behaviour. Use of event reporting (see Section 3.3.2) can close some of the gap, but this makes interpreting PCR values even more difficult. This is a necessary step before individual applications can be assessed and is the problem dealt with in the next chapter.

Chapter 6

From Measurement Logs to System Models

This chapter investigates integrity verification from the perspective of the challenger: can trustworthiness really be established from attestation of an integrity measurement log? Trustworthiness is defined by the Trusted Computing Group as being about how a system *behaves*, but attestation only reports evidence of the *execution integrity* of the platform. This was described in Section 3.1.1 as the *semantic gap problem* and refers to the fundamental difference in what attestation does – provide a history of program execution on a platform – and what a relying party will want to use it for – establishing whether programs on the platform will behave as expected. The difference between these two concepts directly affects how practically useful attestation can be, as reporting execution integrity is a more specific system property than reporting general system behaviour.

Section 6.1 provides an analysis of how execution integrity is established through the TCG-defined chain of trust and why this is inadequate for establishing behaviour. An alternative method is proposed in Section 6.1.3 which does allow the reporting of platform behaviour through the use of *system behaviour models* and the integrity measurement log. Following this, Section 6.2 discusses how to create these models and Section 6.3 and 6.4 explore two alternative implementations, which are then used to describe the TPDMenu program in Section 6.5. Sections 6.6 and 6.7 discuss related concepts and approaches, and finally Section 6.8 concludes.

6.1 Attesting Execution Integrity or Behaviour?

TCG Attestation is about establishing the *execution integrity* of a platform – the identity and integrity of all executable programs – and the process for doing so has been defined in TCG specifications. However, no specification explains how to link this concept to the attestation of platform *behaviour* or *state*. This section first explores the existing attestation process and then identifies how the chain of trust must be augmented to include behavioural information.

6.1.1 Execution integrity reporting

The TCG Attestation approach is well understood, and consists of the following steps.

1. The target platform begins the boot process with a Root of Trust for Measurement (RTM).
2. The RTM and all programs follow the authenticated boot process, building up a chain of trust.
3. A challenger requests the target platform to attest to its execution integrity.
4. The integrity measurement log (IML) and attestation evidence (backed by the Root of Trust for Reporting) are reported to the challenger. The IML contains list of hash values relating to programs and takes the form of a list of lists, one list of hashes per PCR number.
5. The challenger checks that the attestation is valid and roots of trust are trusted.
6. Each reported program hash is checked against a reference 'known good' hash value to ensure integrity of binaries.
7. If all measurements match the reference 'known good' values, and all programs are trusted, then the platform can be trusted.

This process allows the challenger to make sure that unmodified software is running on the target platform – the property of execution integrity – and can be used to enforce a whitelist policy. For client machines, for example, this should be sufficient to check that all software is at the highest patch level, or that no malware is running. This process is elegant and relatively straight forward because the *chain of trust* concept guarantees integrity in a simple hierarchical manner: the integrity of each program relies only on the integrity of the earlier programs.

However, for assurance properties beyond execution integrity a more sophisticated process is required. For example, questions such as 'will the platform keep my data confidentially?', 'is application X still running?' and 'is the system currently being administered by a super user?' rely on platform *behaviour* being known rather than just the integrity of each software component. As a result, the chain of trust concept alone becomes inadequate and must be modified to include a new set of verification steps. Unfortunately, only PCR values can be attested in TCG-defined integrity reporting, and further system properties cannot directly be attested. The question posed by this chapter is whether existing TCG attestation techniques – `TPM_Quote` – combined with a new verification approach is sufficient to report further behavioural evidence.

6.1.2 Behavioural evidence reporting

Evidence of how a platform has and will behave may be more useful to a relying party. They might be able to check to see whether any highly sensitive operations have been carried out or whether a backup process has been run.

The chain of trust can be seen as one piece of evidence about behaviour: the order in which certain pieces of software are loaded on the platform. Knowledge about the behaviour of each piece of software can then be used to predict and trust future behaviour. However, the process for doing this is less well-defined, as each piece of software can behave in many different ways, and the chain of trust only states that a program has be measured, not that it has actually run or performed any particular task [53]. Additional internal information about what each program has done would help provide evidence of a platform's behaviour before the point of attestation, as well as providing an indication of current runtime state.

In order to provide extra information about what each program has done, PCR measurement can be used for *event reporting* (see Sections 3.3.2 and 3.3.3). This is where particular events or significant state changes are recorded into PCRs. For example, in Appendix A a simple ballot box is described which extends the content of each ballot into a PCR value. Similarly, Naumann et al. [147] and Alam et al. [4] extend PCR measurements as part of 'Model-based Behavioural Attestation' to record when a resource has been accessed and how it is used. Events can be represented as the hash of a string of text, and extending them into PCRs has the advantage of making the event attestable and impossible to erase. In the ballot box example, this means that the platform cannot delete any votes after they are originally recorded. This use of PCRs for event reporting starts to provide more information about the behaviour of the platform and can be used to implement a wide range of custom behavioural assurance properties.

Using PCRs for recording both the chain of trust and program-specific events makes the challenger's task of verifying attestations more complicated. As well as having a 'known-good' reference integrity measurement, each program must now also have some 'known good' reference PCRs usage, so that a verifier can tell that if, for example, the string 'error' is extended into PCR 10, the platform should not be trusted. This becomes more difficult because multiple pieces of software may be running at any one time, so integrity measurement logs may contain a set of interleaved event measurements from different programs. Furthermore, programs may run throughout the whole time the platform is switched on. This is demonstrated in Figure 6.1 where the difference between verifying a chain of trust and platform software state can be seen clearly. An example of this is the code running within the Pentium System Management Mode [60]. This may be measured early in the boot process but can still be entered into at any time, making it a relevant part of the overall platform state. Another problem with measuring events, unlike execution integrity, is that the order of is likely to be more important and more liable to change. Event reporting therefore requires a more sophisticated verification process, as well as more information about each program that uses it.

6.1.3 Attesting events

Attestation of events recorded in PCRs requires a more complicated process than the one used in Section 6.1.1 for execution integrity. It is still necessary to check the integrity of each program, but also to identify how each program will behave in combination with the rest of the platform. Behaviour in this context is equated to the events recorded by programs into

PCR values. The following process is required, with new steps given emphasis.

1. The target platform begins the boot process with a Root of Trust for Measurement (RTM).
2. The RTM and all programs follow the authenticated boot process, building up a chain of trust.
3. A challenger requests the target platform to attest to its current state.
4. The integrity measurement log (IML) and attestation evidence (backed by the Root of Trust for Reporting) are reported to the challenger. The IML contains list of hash values relating to programs, *as well as hashes created by programs to mark significant events and state changes*.
5. The challenger checks that the attestation is valid and roots of trust are trusted.
6. Each reported program hash is checked against a reference 'known good' hash value to ensure integrity of binaries. *A trusted reference behavioural model is identified which explains what events the program may extend.*
7. *The models for all programs are combined to create a model for the entire platform.*
8. *This platform model is 'run' against the reported integrity measurement log to establish what* state *the platform is in according to the log.*
   (a) *If the platform is not in any valid state, this means that the platform is exhibiting unexpected behaviour and should not be trusted.*
   (b) *If a valid state of the system model is found, the state and model can be queried from a policy to establish behavioural properties.*
9. *If the platform satisfies the challenger's policy, trust the platform*

This process requires an additional artefact for each program on the attesting platform: a model explaining how it will behave with respect to PCR values. Figure 6.2 gives an intuitive overview of what these models may look like, and Figure 6.3 shows how the process works to establish trustworthiness. It is also necessary to have a way of combining these models together, running them against the integrity measurement log and querying the platform model for behavioural properties. The following sections will investigate the requirements for program models and model running tools as well as details for how they can be used for establishing the current state of a software platform.

6.2 Modelling Programs and PCR Usage

The rest of this chapter investigates how programs which use PCRs to record significant events or state changes can be modelled so that their actions can be interpreted from attestation of an integrity measurement log. There are several open questions, in particular what software

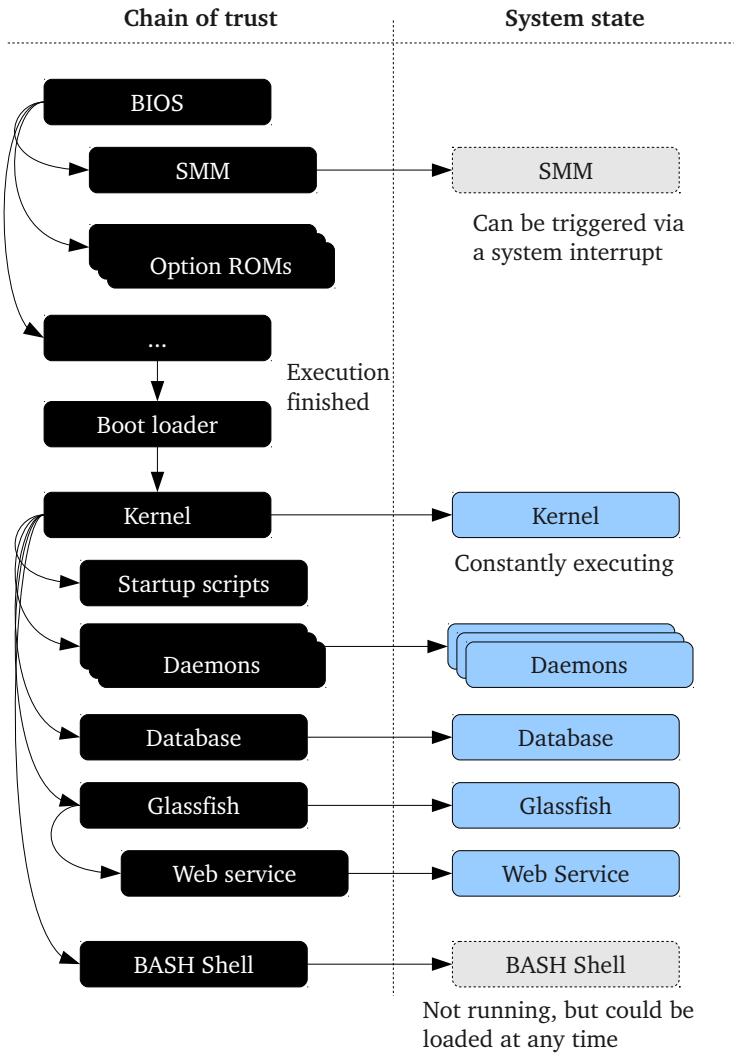

Figure 6.1: Comparing the chain of trust with platform execution state

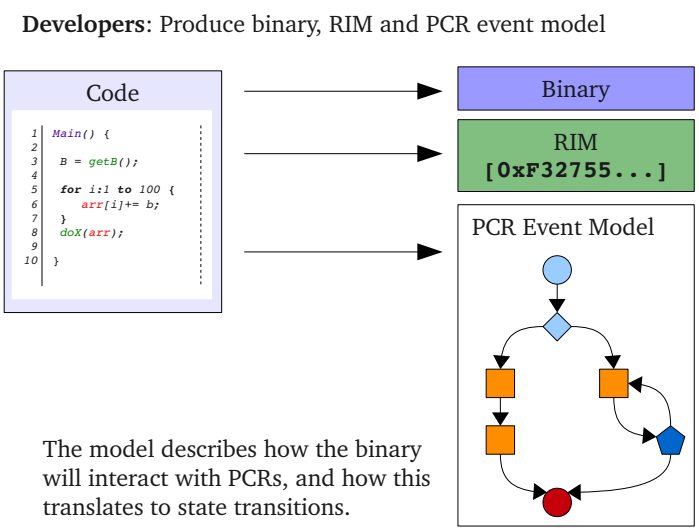

Figure 6.2: Creating models describing program PCR usage

models should consist of, and what technology should be used to check them against an integrity measurement log and identify the current platform state.

As discussed previously in this chapter, program models are used to provide additional information about what a program has done and therefore what state the system is in and whether or not it should be trusted. Examples of the kinds of 'events' that might be recorded include:

- a super-user logging into the platform (and event measured by the operating system shell);
- a document or file being opened (and more usage control properties, discussed in Section 6.7);
- a system backup being run;
- a connection being established to an external service.

A key challenge is that program models should be able to describe smaller pieces of software such as a bootloader, as well as monolithic programs like the operating system. This section identifies design principles for the modelling language, assumptions and limitation of the modelling and verification process and then discusses potential implementation options.

6.2.1 Design principles

The following principles were used to guide the implementations discussed in later sections.

Generality. Program models should be useful for describing any software component on the platform. This means that they must be cross platform and not make too many system-level assumptions. The assumptions that are made should strike the correct balance

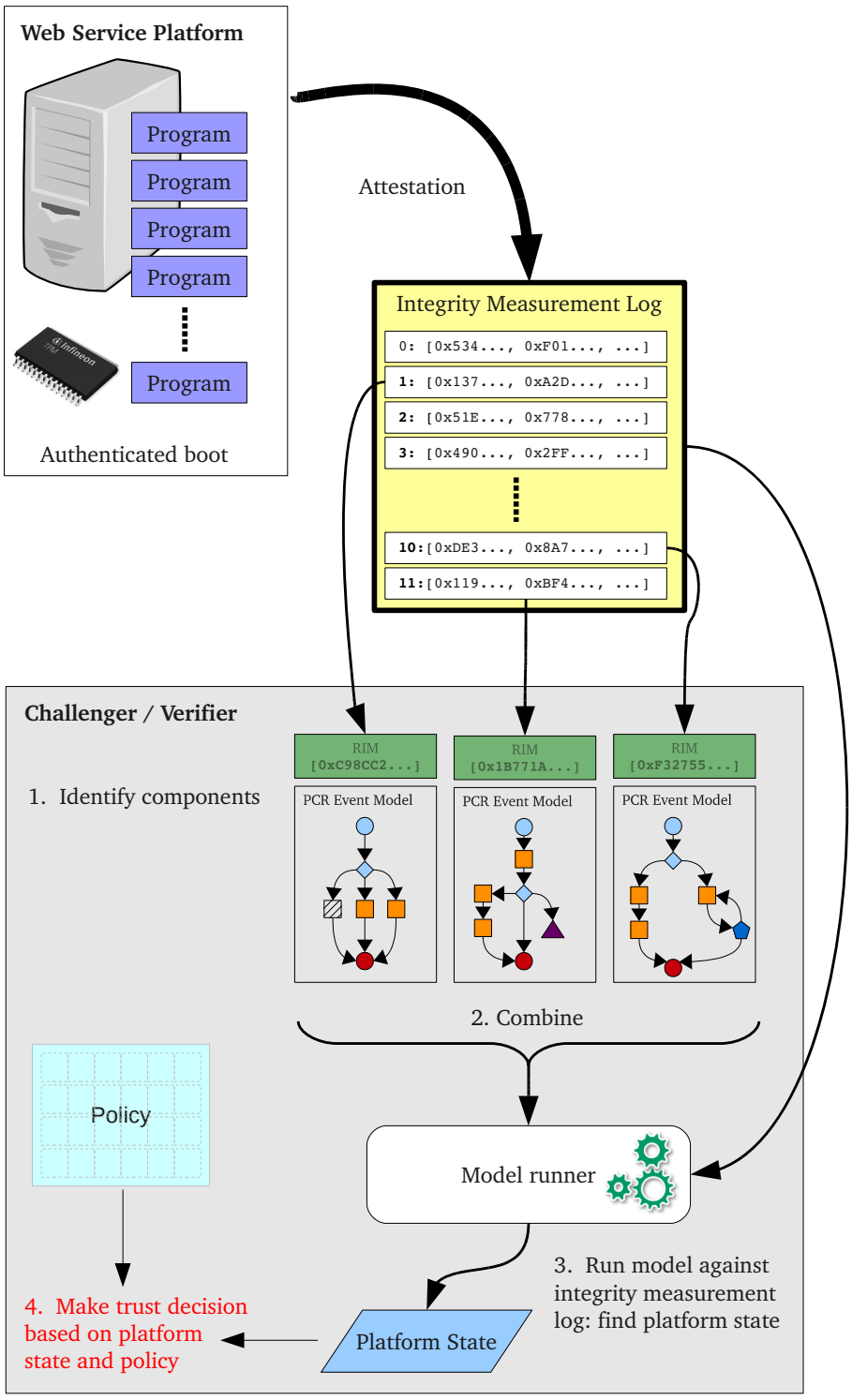

Figure 6.3: An overview of the proposed program modelling approach

between simplicity, to avoid over specification, and yet be platform-agnostic. As a result, some common operating system behaviour may have to be built into the modelling system as opposed to the definition language so that the OS does not require too large a definition.

Uniformity of description. Programs can be written in many languages, including simple shell scripts, and may be modified by textual configuration files. These do not lend themselves well to integrity measurement. Program models must be able to describe these types of programs and files in a common way so that a verifying party only has to understand the mapping from a file to its model, rather than from a file to any particular behaviour.

Composability. Systems should be constructed by composing known program descriptions together. This way a verifying party can use an independent model provided by the developer (or distributor) whom they may trust. For this to work, it should be easy to add program models to an existing system definition without any significant modification.

Support for concurrency. Program models must support concurrency. This adds a great deal of complexity, and platforms wishing to attest will want to avoid too many active, concurrent processes.

Low complexity. The models should be simple to create and interpret, in order to aid usability and verification. This means that program models will need to balance accuracy and simplicity.

TCG compatibility Where possible, TCG concepts should be used, for example, PCR numbers, reference integrity measurements, and measure-before-load. This will also make this work easier to integrate with other trusted computing systems.

Practicality. Ultimately, the goal is to create a practical set of program models that can be used to model a real system. As a result, some of the above principles will face compromise, particularly at the cost of increasing complexity or reducing the generality of the solution. The approach must be capable of describing real programs, and the states they transition between.

Support for hierarchies. Many programs will run alongside (or within) others, and be restricted in what they can do. For example, Java programs run alongside the JRE and are constrained by it. This should be possible to express in the component model.

A program model will naturally be a huge simplification of the program itself. Only PCR usage, passing of control, and event descriptions are of interest. This means that it should be feasible to model large programs such as operating systems.

6.2.2 Assumptions

The use of program models makes several assumptions in principle, even before considering specific implementations. Firstly, each program must have an accurate model which reflects

its real behaviour. Models could be specified incorrectly or be dishonest in what each measurement represents. This means that the model must be trusted and should therefore come from a *trustworthy source*, in the same way that a reference integrity measurement should. Indeed, models could be included with the signed RIM of a piece of software, and when a platform loads a program matching the RIM the correct model could automatically be loaded and used in the verification process.

Another challenge comes with modelling *configuration files*. Configuration files can modify a program's behaviour and may alter the events it will report. They must therefore be part of the system model, and verifiers must make all the same assumptions about the accuracy of a model describing the configuration file that they would about an executable program. However, configuration files are unlikely to come from the same source as programs, as they will be customised for each platform. A solution to this is for the attesting party to provide a complete copy of the configuration file, and for the verifier to generate a model for it. This generation step could be automated in many cases.

The next assumption for a challenger is to decide whether they trust the overall system model (which is now assumed to be built from accurate program and configuration models) and the reported state that the attesting platform is apparently in. This is difficult, but the interpreter can at least say whether or not an unexpected event measurement has occurred, which might imply runtime compromise or an error. However, a policy will still need to be defined for interpreting the reported system state. The system model may define some states which are explicitly untrustworthy (for example, if the user 'root' logs-in and starts a new terminal). However, some of these policies could be context-specific and therefore defined only by the challenger. The problem of creating policies is not considered in this chapter.

There is still potential for runtime attack as if any program is exploited at runtime they may deviate from the model. However, if the event reporting is designed well then any attack will be made visible by an unexpected state change in the event log. Because these cannot be modified later, the attack can be spotted and subsequent actions can be marked as untrusted. Furthermore, this approach can work in combination with a runtime agent in order to provide defence in depth. If the agent is compromised, then the event log may provide evidence of this. If the running application is compromised, the runtime agent may alert the relying party. Either way, this approach will automate the *verification* process so that a decision about trustworthiness is easier to make.

Finally, there will be more assumptions made by the specific model and implementation. These are discussed in later sections.

6.2.3 Limitations: predicting the future

The general approach described in this chapter can be used for verifying an integrity measurement log against an expected platform model. However, there are some significant limitations which should be established. It is not possible to use these models to predict potential *future* behaviour. This is because any program might be loaded into memory and executed: the integrity measurement log only provides evidence of what *has* happened, rather than what *will*.

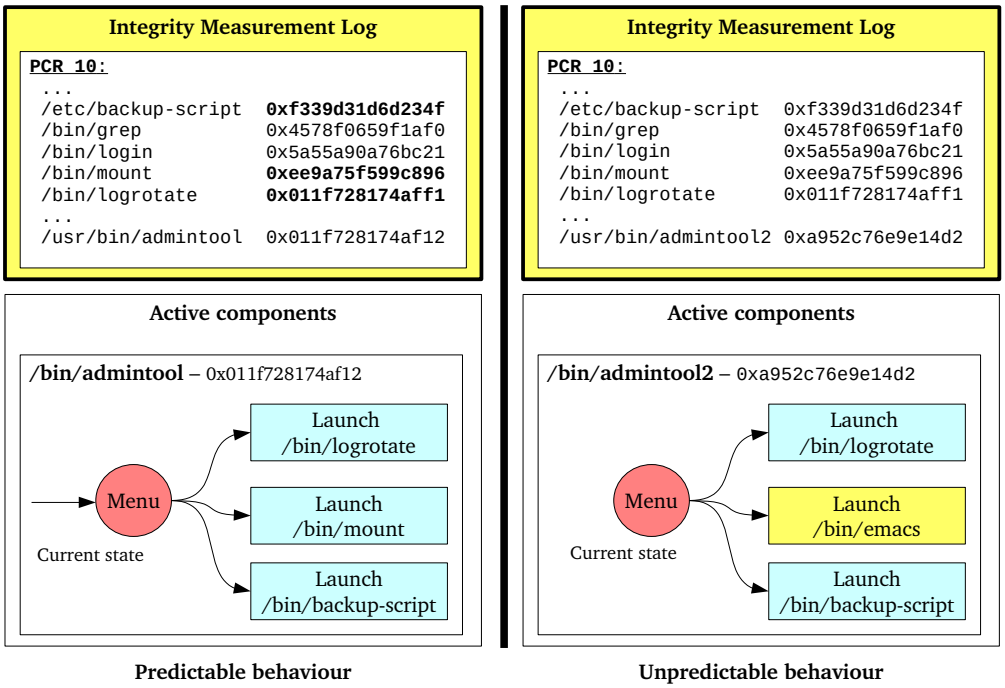

Figure 6.4: An example of the conditions required for predicting future behaviour based on an attestation

A new executable could be malicious, or have been altered in an unexpected way. However, by making some assumptions, it is possible to have a degree of assurance as to future behaviour.

There are two conditions where this might be true. The first is if one of the running programs allows for an already-measured component with a given file name to be executed. Because the file name and hash are already in the log, the challenger knows that that program *may* be run in the future. If all possible future states of the active model satisfy this constraint, then the range of all possible actions can be constrained to just these — assuming, of course, that the filename-hash mapping is preserved and cannot change. This could be enforced by the operating system. It is also assumed that the active model contains all possible events, including any interrupt handlers, device drivers and running processes. An example of this constraint can be found in Figure 6.4.

The second condition is if a secure-boot style system is followed. Using the example in Figure 6.4, if the menu was also programmed to only execute those files *if* they had a certain hash, and the menu was trusted to enforce this, then future actions are similarly constrained. This is unlikely to be the case for most applications, but is a useful alternative when trying to limit the range of possible future behaviour.

6.2.4 Implementation options

Having defined the assumptions and goals of an event attestation model the next two sections describe alternative implementations. Both have advantages but neither is a perfect solution.

For the sake of clarity all models shown in this chapter do not extend hashes, but text values into PCRs. These would need to be converted in a real implementation.

6.3 CSP Program Models

Communicating Sequential Processes (CSP) is a process algebra commonly used to describe interacting concurrent systems. It allows the definition of individual components which can be composed together to produce larger system models.

A full explanation of CSP can be found in [174]. Processes are defined by name and make a series of communications before either stopping (taking the behaviour of process 'STOP' which is defined as a process that never communicates) or behaving like another process. For example, process P communicates a and then behaves like Q. Process Q communicates b and then never communicates again:

$$P = a \rightarrow Q$$
$$Q = b \rightarrow STOP$$

Arrows ($\rightarrow$) show the sequence of events. Processes may be composed in parallel and must synchronise on any communications they share. These are defined explicitly in the composition, for example $P \llbracket a, b \rrbracket Q$ shows process P and Q must synchronise on communications a and b. External choice is shown with a square ($\Box$).

Messages can be communicated between processes through channels which have *inputs* and *outputs*. In the diagrams in this chapter, inputs are shown with a question mark and outputs are shown with an exclamation mark. For example, the TPM process in Figure 6.5 shows the TPM waiting to synchronise on channel *extend*, where it receives a message into object x and then outputs the same value on channel *tpmextend*. Multiple input and outputs are separated using the same question mark, dot or exclamation mark. If a specific message is defined for the input, the process will only synchronise if the right output on that channel is given by another process. For example, the *PCRLOG*1 process in Figure 6.7 synchronises on two values, the first of which must be 1 and the second must be *bios*.

6.3.1 Using CSP to verify integrity measurement logs

Assuming that all program models are defined using CSP, the FDR2 tool [64] can be used to check a system model against a trace of actions through *trace refinement*. Trace refinement [174] can confirm that a measurement log could have been produced by the system model. In the model defined in Figure 6.5 this can be done by converting the measurement log into a single sequential process of *tpmextend*!*name* actions and then checking that the system model is a refinement of this measurement log model. In other words, the system model should have been able to produce this trace. The ProBE tool [49] can then be used to visualise the state of

the model after the trace has been run.

CSP provides several concepts which are immediately applicable to integrity measurement. The behaviour of the system given this measurement log (*LOG*) has been reported is defined as the 'afters' (*SYSTEM*/*LOG*) of the trace. This is useful for establishing what possible events could occur given the current system state after the trace. For example, after an administrative intervention, will the platform still accept user input? CSP defines *refusals* and *failures* which fit well for specifying these policies. A *refusal set* is a set of events that a process will *never* be able to accept anything from. A *failure* is a pair (s, X) where s is a trace of process P and X is a member of the set of refusals of P/s. In other words, after the trace s, process P is unable to ever accept any of the events in set X.

CSP can satisfy many of the properties listed in Section 6.2.1. It supports non-determinism and parallel processes. It is suitably abstract and can model any application. Configuration files can be specified as additional processes. Indeed, the example in Figure 6.10 shows that configurations can be composed with applications, so long as they use known message types. TCG concepts are also easy to encode (the idea of PCRs, a single TPM, and a measurement list fit well). For verification, individual logs are composed together, as shown in Figure 6.7.

The following examples demonstrate how CSP can model parts of the boot sequence. In Section 6.3.4 validation of these models against a measurement list is considered, and Section 6.3.5 discusses outstanding problems with the implementation.

6.3.2 Example platform model

An example simplified system model can be seen in Figure 6.5. It shows an authenticated boot process consisting of the TPM, BIOS, Bootloader, IMA Linux operating system, the TTY and then the undefined process APPS which is a place-holder for any other programs also on the system. The TPM is a constantly running process which receives messages on the `extend` channel, containing two arguments: the PCR number and the value to extend. When it has finished, it responds with a `finishextend` message with the same arguments, making it a synchronous process. The BIOS is the root of trust, extending itself and then calling for the bootloader to be extended and executed. In turn, the bootloader extends `ima`, the operating system. The IMA process runs continuously, waiting for either `launchreq` messages – which are requests to launch application `x` – and `extendreq` messages which are arbitrary requests to extend a value to PCRs. The model also corresponds to how the real IMA system works, caching program measurements, so that only new programs are measured into a PCR. Each application begins with a `launchreq!APP-NAME`, so that measurements are triggered. For example, the TPDMenu process shown in Section 6.5 would be one of the processes in `Apps`.

The final SYSTEM process is the parallel composition of the other processes, forcing them to synchronise on certain channels. The SYSTEMH process is the SYSTEM process with internal communication hidden, so only the TPM processes `tpmextend` communication is visible, allowing it to be compared to a integrity measurement log process (see Section 6.3.4).

SystemBoot

channel
extend, *tpmextend*, *finishextend*

channel
channellaunch, *launchreq*, *extendreq*, *hasextended*

process
$TPM = extend?p?x \rightarrow tpmextend!p!x \rightarrow finishextend!p!x \rightarrow TPM$

process
$BIOS = extend!1!bios \rightarrow finishextend?1?bios \rightarrow$
$extend!2!bootloader \rightarrow finishextend?2?bootloader \rightarrow BOOTLOADER$

process
$BOOTLOADER = extend!7!ima \rightarrow finishextend?7?ima \rightarrow IMA(\langle\rangle)$

process
$IMA(s) = launchreq?x \rightarrow CACHER(s, x) \;\Box$
$extendreq?x \rightarrow extend!10!x \rightarrow finishextend?10?x \rightarrow hasextended!x \rightarrow IMA(s)$

process
$CACHER(s, x) = \text{if } elem(x, s)$
$\text{then } launch!x \rightarrow IMA(s)$
$\text{else } extend!10!x \rightarrow finishextend?10?x \rightarrow launch!x \rightarrow IMA(s \frown \text{seq}\, x)$

process
$SYSTEM = (TPM \;[\![\, extend, finishextend \,]\!]$
$(BIOS \;[\![\, launchreq, launch, extendreq, hasextended \,]\!]\; (TTY \;|||\; APPS)))$

process
$SYSTEMH = SYSTEM \setminus$
$\{extend, finishextend, launch, launchreq, extendreq, hasextended\}$

process
$TTY = launchreq!shell \rightarrow TTY$

Figure 6.5: CSP model of platform boot and IMA Linux

6.3.3 Example script model

The kernel start-up process was modelled in order to test how well CSP could describe scripts and configuration. On Linux this involves reading scripts from the subdirectories of `/etc/rc.d` directory and executing them in order. Many are shell scripts. Figure 6.6 gives an example of how this can be modelled. It is worth noting that BASH is particularly difficult to model. As will be discussed in Section 6.3.5, the interaction between the script and interpreter is hard to define in a simple manner. The '`...`' sections show where the script behaviour would need to be specified.

StartupScripts

$scriptlist = \langle script1, script2, script3 \rangle$

process
$RCD = launch?rcd \rightarrow RCDINNER(scriptlist)$

process
$RCDINNER(\langle\rangle) = STOP$
$RCDINNER(\langle s \rangle \frown t) = launchreq!s \rightarrow RCDINNER(t)$

process
$SCRIPT1 = launch?script1 \rightarrow launchreq!bash \rightarrow ... \rightarrow STOP$

process
$SCRIPT2 = launch?script2 \rightarrow launchreq!bash \rightarrow ... \rightarrow STOP$

process
$SCRIPT3 = launch?script3 \rightarrow launchreq!bash \rightarrow ... \rightarrow STOP$

Figure 6.6: CSP model of platform startup scripts

6.3.4 Verification process

The purpose of verification is to establish that the attesting system used PCRs in the manner defined by the program models. If verification fails it may imply:

- a failure of a program to properly follow measure-before-load, resulting in unmeasured code being able to modify the TPM;
- an error at runtime, possibly due to a bug or runtime attack;
- an unexpected program (without trusted model) being executed; or
- the modification of a program, resulting in a different hash value.

However, if verification *succeeds*, this implies that the used program models *may* be a good model for the system. However, verification does not prove that a runtime attack has not occurred, or that a program *has* obeyed the measure-before-load policy. If the model has ambiguities, or if the program has undocumented behaviour, there can still be difficulties. Verifying the PCR usage models is just one necessary step in the larger assurance process.

Once models have been defined, the following steps should be used to verify the measurement log. First, the appropriate models are selected based on the integrity measurements shown in the log. The CSP models are then composed together into one FDR script. The attesting platform's measurement log is converted into a process showing the TPM's actions. Next, FDR2's trace refinement checker is used to compare the log against the system model. The result of this shows whether or not the platform has behaved as the model specifies. An example of this is shown in Figure 6.7 for verifying the system in Figure 6.5. It shows that the measurement log refines the system model, which means that the traces defined in the log are

a subset of the possible traces defined in the model. In other words, the log shows that one of the possible ways in which the model could have transitioned has been followed. In order to inspect the resulting platform state, the ProBE tool can be used to identify the applications that are still running, and the current process model. Many further checks are possible but are left for future work.

Note that the `tpmextend` measurement values are all names rather than hashes: converting to hashes requires an additional step before model execution. The actual cryptography and hashing functions are also not modelled in this approach.

LogVerificationExample

channel
$extend, tpmextend, finishextend$

process
$PCRLOG1 = tpmextend.1.bios \rightarrow ...$

process
$PCRLOG2 = tpmextend.2.bootloader \rightarrow ...$

assert
$SYSTEMH \sqsubseteq_t (PCRLOG1 \,|||\, PCRLOG2...)$

Figure 6.7: CSP log verification example

6.3.5 Problems

There are several problems with using CSP. The first issue is tool support. It is difficult to observe the resulting platform state after the integrity measurement log have been verified. FDR2 is also a problem – in the examples in Figure 6.5, FDR2 cannot check trace refinement due to a state-space explosion caused by the caching process. Removing this allows for refinement checking, but makes the model less accurate and implies that this approach will not scale well for larger models.

The second issue is that models of configurations and scripts are far from intuitive. Configurations are *not* equivalent to processes, but must be modelled in this way. Furthermore, scripts present difficulties. Should the behaviour be described in the model for the script file, or as an argument for the interpreter? The script is not the actual process performing the actions, but it does make sense to encapsulate its behaviour in this way. As the script may not know which version of the interpreter will be used, how does it state that, for example, BASH might be expected to run? A similar problem is found with managed code. The interaction between the JVM and Java classes, for example, is quite complicated, but CSP does not provide any obvious language features for encapsulation or containment of processes.

The lack of process hierarchies also means that there is no sense of ownership: the process representing the OS cannot kill other processes without explicit design, adding a great deal

of complexity. This problem is apparent when trying to model the idea of a user logging in, starting processes, and then logging out. This should result in the run processes being killed, but this is difficult to model. There are other minor issues when describing applications in this style – the change in semantics means it may be harder for developers to describe their programs. Furthermore, FDR2 does not support passing arbitrary strings as messages, which is necessary for implementing Behavioural Attestation [4] and the loading of arbitrary files. Directories also have to be modelled as files. This means that the `/etc/init.d` directory, which is normally opened to find individual scripts, must be treated as one process. A better abstraction would allow statements such as 'load each process from list X.' Finally, it is difficult to compose applications which are unaware of each other in a sequential manner. This results in an interleaved message-passing system which is counter-intuitive for describing sequential actions.

Overall, the CSP approach is extremely promising and provides an elegant way to describe some applications. However, further investigation of alternatives is required to overcome some of the problems. The next section investigates whether a custom program description implemented in Prolog would be a better solution.

6.4 Implementation in Prolog

The logic programming language Prolog can be used to implement both the description language and the modeller. A full background of the language is not provided in this thesis, but a good Prolog tutorial can be found at [100]. Prolog and CSP are very different: Prolog is not a process algebra but a programming language, and therefore requires the development of a custom process model and language for describing programs. However, it does inherently support non-determinism, which is useful for matching the integrity measurement log to a system model, as several non-deterministic cases are possible. For example, when the same program is run multiple times concurrently on the platform, both of which can extend the same measurements.

There are several immediate advantages over modelling in CSP. A Prolog implementation must define its own process model, with custom-made data types for the programs and processes involved, and can therefore be more intuitive to read and design than a CSP model. As a result, programs can define internal state and how they transition from one state to another. They can also be hierarchical, and communicate in custom ways. Furthermore, the Prolog system that was developed can both validate measurement lists *and* produce plausible lists based on a given set of programs. This may be useful during system or program design.

Configuration can also be modelled as a special case. In the implementation described in this section, configurations *override* the behaviour of pre-defined applications. This makes sense for many situations, such as when a feature can be turned on or off. To prevent impossible configuration settings, the applications can define precisely which behaviour is overridable. An example of overriden states can be seen in the JRE model in the following section.

6.4.1 Example models

The following models demonstrate the boot process of a platform, similar to the CSP models discussed previously. Programs are modelled as transitioning state machines which can be interleaved to model parallelism. Programs have a label and start-state, and each state consists of an application name, state id, priority (lower is better, 0 is the minimum), measurement to extend to a PCR (may be empty), and a next-transition function. Measurements are shown as strings for readability, but must be converted to hashes on a real implementation. Transitions are either to internal states (`newstate`) , the loading of a new application, or can be operators such as `parallel` or `choice`. An `app` transition loads a new application, with the second part of the tuple being arguments to the program. This program and its arguments are measured automatically. An advantage of this modelling approach is that new types of transitions can be defined so that application-specific concurrency or transitions can easily be implemented. The models shown in this section demonstrate the flexibility of the Prolog-based approach.

The system model starts with a definition of each component on the platform using the **`programme`** clauses. The first argument is the program name, and the second is the set of initial transitions. In this case, there are 12 programs, including the BIOS, bootloader, operating system (Linux) and various applications:

```
% Programmes are defined with a unique name and initial state.
programme(bios,             [newstate(bios-init)]).
programme(bootloader,       [newstate(bootloader-init)]).
programme(linux,            [newstate(linux-init)]).
programme(cron,             [newstate(cron-init)]).
programme(webservice,       [newstate(ws-init)]).
programme(tpdmenu,          [newstate(tpdmenu-init)]).
programme(psapp,            [newstate(psapp-init)]).
programme(whoapp,           [newstate(whoapp-init)]).
programme(backup-script,    [newstate(backup-init)]).
programme(logrotate-script,[newstate(logrotate-init)]).
programme(bash,             [newstate(bash-init)]).
programme(jre,              [newstate(jre-init)]).
```

The initial boot sequence is given below. The BIOS begins, has the chance to perform some measurements (in this case, `'bios init'`), and then loads the bootloader. Recall that each `app` transition will *automatically* generate a measurement, so this does not need to be given in the model. The bootloader is given the configuration file `grubconf` as an argument. This contains the state `bootloader-stage1` which transitions to the operating system.

```
% The initial boot sequence: BIOS, and grub bootloader.
% Each programme can have multiple states, each state consists of
% The programme name, state name, priority, measurement and next
% transition
state(bios, bios-init, 0, 'bios init', app(bootloader,[grubconf])).
state(bootloader, bootloader-init, 0, 'bootloader init',
```

```
  newstate(bootloader-stage1)).
state(grubconf   , bootloader-stage1, 50, 'grub config',
  choice([app(linux,[linux-passwd, linux-initd])])).
```

The operating system, in this example, also includes the loading of device drivers or modules. To simulate the potential uncertainty in the order that these are loaded, modules are run in parallel. Although in this example they exist as *states*, they could also be implemented as separate components. The order of events is: the kernel starts, three drivers are loaded, and then services and the TTY process start in parallel. Each kernel module extends a 'loading' message, which in a real system would be replaced with the hash of the kernel module.

```
state(linux , linux-init , 0, 'linux started',
  dothen(
    parallel(
      [newstate(linux-vid-drv),
       newstate(linux-kbd-drv),
       newstate(linux-usb-drv)]),
    newstate(linux-finished-init) )).
state(linux , linux-vid-drv, 0, 'loading video driver', end).
state(linux , linux-kbd-drv, 0, 'loading keyboard driver', end).
state(linux , linux-usb-drv, 0, 'loading usb driver', end).
state(linux , linux-finished-init, 0, 'linux finished INIT',
  parallel([newstate(linux-svc),newstate(linux-tty)])).
```

The TTY process allows for multiple concurrent logins as either root or the user 'PDUSER.' The option of users and shells is defined in the `linux-passwd` states, modelling the `/etc/passwd` file in Linux. In this example, root does not have a shell defined, and the login process has been modified to extend two values: `LOGIN` on login and then the user name.

```
% The login and terminal process.  Concurrent sessions are possible, and
% either a root user or a menu-based shell user may log in.
state(linux , linux-tty, 0, 'TTY', parallel([newstate(linux-login)])).
state(linux-passwd, linux-login, 0, 'LOGIN',
  choice([ newstate(tty-pduser), newstate(tty-root)])).
state(linux-passwd, tty-pduser, 0, 'PDUSER', app(tpdmenu,[])).
state(linux-passwd, tty-pduser, 0, 'ROOT', end).
```

The following examples show the service daemons started by Linux. These include CRON, for scheduled tasks, and a web service stub. The CRON program regularly checks the CRONTAB file, which is a configuration file modelled in the `crontab` states. This runs the `backup-script` and `logrotate-script`, both of which run inside Bash. Again, many additional measurement states have been added to these models to show the additional behavioural information that could be gained from this approach.

```
% The startup scripts.
state(linux-initd , linux-svc, 0, 'start services',
```

```
  parallel([app(cron,[]),app(webservice,[])])).

% The CRON background task
state(cron , cron-init, 0, 'CRON STARTED', newstate(cron-loadconfig)).
state(cron , cron-loadconfig, 0, 'loading cron config',
  dothen(loadconfig(crontab), newstate(cron-start))).
state(crontab, cron-start, 0, 'CHECKING CRONTAB',
  parallel([  app(backup-script,[]), app(logrotate-script,[]) ])).

% Some example maintenance scripts
state(backup-script, backup-init, 0, 'starting backup',
  app(bash, [backup-script])).
state(backup-script, bash-script, 0, 'starting backup in bash',
  newstate(cron-start)).
state(logrotate-script, logrotate-init, 0, 'starting logrotate',
  app(bash, [logrotate-script])).
state(logrotate-script, bash-script, 0, 'starting logrotate in bash',
  newstate(cron-start)).

% Bash.  Note that this just calls the script
state(bash, bash-init, 0, 'starting bash', newstate(bash-script)).

% A stub for a Java web service
state(webservice , ws-init, 0, 'WS STARTED', app(jre,[ws-java])).
```

The next model shows a Java runtime environment and stub web service application. The JRE loads a number of libraries when it start up, and then gives empty states `jre-loadclasses` and `jre-app-start` which can be overriden by the Java program given as an argument. In parallel with running the application, the garbage collector is shown to run constantly, and perform a measurement when it does so. The `ifopenstates` transition will call the next transition only when the application itself has remaining states to transition to.

```
% The JRE with libraries, a garbage collector, and stub web service.
state(jre, jre-init, 0, 'JRE start',
  dothen( parallel(
      [newstate(jre-measure-sys1),
       newstate(jre-measure-sys2),
       newstate(jre-measure-sys3)]),
    dothen(newstate(jre-loadclasses),
      parallel([newstate(jre-app-start),
                ifopenstates(newstate(jre-gc)) ]) ))).
state(jre, jre-measure-sys1, 0, 'jre-libverify.so',end).
state(jre, jre-measure-sys2, 0, 'jre-libjava.so',end).
state(jre, jre-measure-sys3, 0, 'jre-libzip.so',end).
state(jre, jre-gc, 0, 'garbage collect', ifopenstates(newstate(jre-gc))).
state(jre, jre-end, 0, 'JRE shutdown', end).
```

```
state(jre, jre-loadclasses, 100, '', end).
state(jre, jre-app-start, 100, '',end).

% These are examples of overriden states, in this case to show how
% a Java application can be started.
state(ws-java, jre-loadclasses, 50, 'service.jar', end) .
state(ws-java, jre-app-start, 50, 'WS Java loaded', newstate(...)).
```

6.4.2 Running the model

The algorithm for executing the model is shown in full in Section B.2 but is summarised as follows. The `iterate_start(_,_,_,_)` statement is defined, which takes a starting program, list of configuration files, measurement list, and final platform state. It will return yes or no, depending on whether the given platform state and measurements are consistent. This is calculated by treating the final platform state and measurement list as *outputs* of running the given starting program.

Program models are executed by maintaining a list of currently-running programs, initially containing the first argument to the `iterate_start` function. A program is chosen non-deterministically from this list, and allowed to output a measurement, based on the configuration of the states it can transition to. This may alter its internal state, or result in a new application being run and added to the list. This process continues while new transitions are still possible.

6.4.3 Verification

To verify a measurement log, the following steps are needed. First, all the program models are given as facts in the Prolog script through **`programme`** and **`state`** clauses. Having defined the behaviour of all programs, the following statements need to be executed by the Prolog interpreter:

```
iterate_start(bios, [], [ ... measurement list ...], PSTATE).

% iterate_state function, arguments:
% 1 - root of trust component,
% 2 - any initial configuration options for the first programme,
% 3 - integrity measurement log,
% 4 - resulting platform state.
```

This will return all possible PSTATE variables which could explain the measurement list, assuming the BIOS program started first without any new configuration. Alternatively, if the measurement list is left as an ungrounded variable (MLIST), the model can be used to generate all possible measurement lists and states:

```
iterate_start(bios, [], MLIST, PSTATE).
```

Measurements are either shown as `measure(App,Configs)` called for application measurements, or just text strings. A program state takes the form of a tuple, with program id (generated at instantiation), program name, state table, parent process id, and the set of possible next transitions. An example measurement list and platform state based on the models given in the previous section is shown below.

```
MLIST = [
measure(bios, []),
'bios init',
measure(bootloader, [grubconf]),
'bootloader init',
'grub config',
measure(linux, [linux-passwd, linux-initd]),
'linux started',
'loading video driver',
'loading keyboard driver',
'loading usb driver',
'linux finished INIT',
'start services',
measure(cron, []),
measure(webservice, []),
'TTY',
'LOGIN',
'PDUSER',
measure(tpdmenu, [])
]

PSTATE = [
(t38, tpdmenu, [...], t35, [newstate(tpdmenu-init)]),
(t37, webservice, [...], t35, [newstate(ws-init)]),
(t36, cron, [...], t35, [newstate(cron-init)])]
```

The PSTATE variable shows that three processes are still able to run: the tpdmenu program, a web service and the cron daemon. The measurement list records all the measurements made in the boot process of the platform, including three kernel modules, the operating system, and more.

In order to simulate the caching behaviour of IMA Linux, the output of the `iterate_start` function must be fed into another function to remove duplicate `measure` entries. This is inelegant, but easy to implement. The alternative would be to have more complex program definitions, which allow for internal variables to be kept track of. For the majority of programs, however, this is unnecessary. An additional step is also required. All 'measure' entries must be replaced with the actual SHA1 hash values of the executables. This can be implemented as a simple look-up table. Other event strings, such as `'start services'`, will also need to be replaced with their hash, or might be removed.

Several explanations are produced which show what state the platform could be in. These should all be iterated through to make sure none contain an untrustworthy state.

6.4.4 Problems

Unfortunately, the increased flexibility of Prolog has an attached cost. Models are more complicated, as is the interpreter. This makes it more likely that an error will be made in a model, reducing accuracy and trustworthiness. Furthermore, this makes it harder to standardise models, and more difficult to assess whether a model is a reasonable approximation of the component. Part of the added complexity comes from the fact that concurrency must be programmed in, unlike with CSP. Finally, there are issues with verification. Prolog performs a depth-first search, and the model implemented here can go into infinite loops, endlessly extending the measurement log rather than taking a different path. While this could be fixed to some extent, it does spoil the elegance of the solution. By having to manually implement a breadth-first search, many of the benefits of using Prolog are lost.

6.5 The TPDMenu Shell

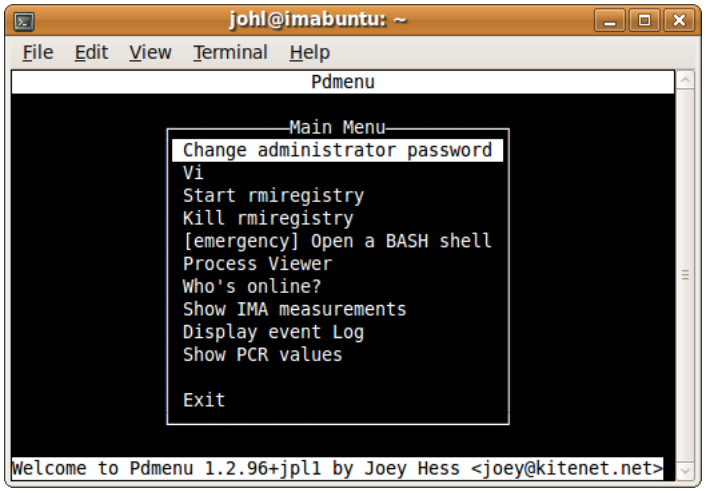

Figure 6.8: An example instance of PDMenu.

One of the problems discussed in Section 6.2.3 is of using attestation to have trust in *future* behaviour. This can only be achieved if the attested processes all have well-defined future states and no new executable has the chance to be launched at any time. This is not possible when user-input or administration is allowed at a Unix shell, as is often the case. One alternative is to provide limited administration through a pre-defined *menu*, for example, menus used by application servers to configure web services. However, these tend to be part of a larger middleware stack, and work in Chapter 5 has avoided putting these on attested platforms.

A light weight alternative is a menu-based command shell. This restricts the administrative

user to just the commands presented on the menu. If set as the user's default shell, it should be infeasible to break out of. PDMenu [89] is one such shell, which has a simple configuration file defining the menu commands that can be executed. PDMenu was modified to support *integrity measurement* so that every command string is measured before being run. As the kernel still measures the actual binaries, this means that all actions are measured before being run. An example of the menu configuration file can be found in Figure 6.9, and an example of the menu itself in Figure 6.8. In order to guarantee that this shell is running, and not any other, the kernel's login program will measure the shell launched.

Menu-based shells can also be used to implement 'break-the-glass' style policies [62]. These allow emergency actions to be performed, so long as they are reported or result in additional constraints. The menu item '`exec:[emergency] Open a BASH shell`' given in Figure 6.9 is an example of this. A bash terminal can be opened, but results in PCR 12 being extended. This will be reported in attestations, and data can be sealed to the value of PCR 12 so that it is made unavailable after this action. To maintain a record of this event *after* system reboot, TPM counters can be used in a similar manner to the ballot box implementation in Appendix A.

An example of a TPDMenu CSP model is shown in Figure 6.10, complete with a modification to the `/bin/login` program at the time in which the `/etc/passwd` file is read, and a shell is started. In this case PCRs are invalidated when a BASH shell is run. Note that this model demonstrates *configuration*.

```
menu:main:Main Menu
  exec:Change administrator password:p:passwd
  exec:Vi::vi
  exec:Start rmiregistry:p:rmiregistry &
  exec:Kill rmiregistry:p:killall rmiregistry
  exec:[emergency] Open a BASH shell::
       ./jtt.sh pcr_extend -f emergency.txt -p 12; bash
  exec:Process Viewer:truncate:ps aux
  exec:Who's online?:truncate:echo "These users are online:";w
  exec:Show IMA measurements:display:cat /sys/kernel/security/ima/ascii*
  exec:Display event Log:display:cat ./menulog.txt
  exec:Show PCR values:truncate:./jtt.sh pcr_read
  exit:Exit
```

Figure 6.9: An example PDMenu terminal configuration file

A Prolog version of the TPDMenu shell is defined below. It is given the `pdmenurc` argument at login, simulating the real behaviour of the shell and its configuration file. In this example, the configuration file allows the user to run the `ps` and `who` commands.

```
% The menu-based shell.
state(tpdmenu, tpdmenu-init, 0, 'TPDMENU START',
  choice([newstate(tpdmenu-exit), newstate(tpdmenu-menu)])).
state(tpdmenu, tpdmenu-exit, 0, 'TPDMENU EXIT', end).
state(tpdmenu, tpdmenu-menu, 0, 'LOADING MENU',
  dothen(loadconfig(pdmenurc), newstate(tpdmenu-menu-config))).
```

```
% Menu configuration: choice between two options.
state(pdmenurc, tpdmenu-menu-config, 0, 'TPMENU CHOICE',
  choice([  app(psapp,[]) , app(whoapp,[])])).

% Some example menu options
state(whoapp, whoapp-init, 0, 'who launched', end).
state(psapp, psapp-init, 0, 'ps launched', end).
```

TPDMenu is an example of a program which has a simple model and can make integrity measurement more meaningful. By modelling the configuration and the application, it is clear that only certain actions are possible and therefore that the platform may be trusted for the duration of this boot. The configuration file is easy to parse, and effectively works like a whitelist.

TPDMenu

process
$TTY = launch?shell \rightarrow (launchreq!tpdmenu \rightarrow STOP)$
$\square$
$(extendreq!invalidatepcr$
$\rightarrow hasextended.invalidatepcr$
$\rightarrow launchreq!bash \rightarrow STOP)$

process
$TPDMENU = launch?tpdmenu \rightarrow TPDMINNER$

process
$TPDMINNER = extendreq!started \rightarrow hasextended?started \rightarrow TPDMCONF$

process
$TPDMQUIT = extendreq!finished \rightarrow hasextended?finished \rightarrow TPDMENU$

process
$TPDMCONF = extendreq!who \rightarrow hasextended?who \rightarrow launchreq!who$
$\rightarrow TPDMCONF$
$\square$
$extendreq!ps \rightarrow hasextended?ps \rightarrow launchreq!ps$
$\rightarrow TPDMCONF$
$\square$
$TPDMQUIT$

Figure 6.10: CSP model of the TPDMenu menu-based shell

6.6 Discussion

Problems encountered with Prolog and CSP implementations have highlighted some interesting further issues. It seems clear that the CSP model works well for relatively simple processes without hierarchical behaviour. The tools can scale well until memory is required, making them appropriate for early on in the boot process. The Prolog system is much more flexible, and more suited to describing complicated applications. By using *both* tools, many of the problems can be reduced. However, the overall conclusion of these attempts is that the problem would be greatly reduced if the amount of software was also reduced. A smaller operating system and less middleware would aid in verification and modelling. There are some further issues and design decisions which are discussed in the rest of this Section, including where to store hash values, how to integrate modelling into the build process, dynamic roots of trust, the use of a measurement agent as an alternative, and more.

6.6.1 Program hashes: Part of the model?

One of the most difficult design choices in modelling has been where to include binary hashes of programs. These are recorded in the RIM of the application, but should they also be in the model? The problem is that the program responsible for measuring the hash is, in fact, the program that loads the binary, not the binary itself. Therefore, it makes sense to include the hash in the preceding component. Unfortunately, this is a bad idea in terms of encapsulation: the hash is something known by the developer of that piece of software, and as they provide a RIM anyway, it should arguably be at the start of its own model.

This problem can be partially solved by not using hashes in the model, but matching them later on. For example, the `BIOS` process measures the place-holder string 'bootloader' rather than a hash. When the model comes to be validated, the actual hash in the IML is replaced. The hash is still important and must be processed – it validates which bootloader model to use. In more complicated scenarios, such as IMA, an alternative solution was to make each application wait to be measured before allowing it to continue.

6.6.2 Integration into the build process

Creating program models is a tedious task and there is no guarantee of their accuracy. A more appealing approach is to *generate* models from the source code of the application. In this way, developers can release a trusted RIM of the application, along with a model, requiring little additional development effort.

There are several promising approaches for implementing this idea. An aspect-oriented system could separate the use of PCRs from the rest of the system, and then publish a model based on just these aspects. Win et al. [237] have already discussed separating security functionality through aspects. Alternatively, a model-driven approach such as that taken by Booster (see Section 2.4.4) would allow parts of the original model to be reused for specifying PCR usage. Other systems such as CHSM [120] allow programs to be written with their state chart

in mind.

6.6.3 Modelling further actions and constraints

To simplify models only PCR events have been described. However, several other actions could potentially be included. The TPM Quote operation, for example, could be part of the platform description. This would add further legitimacy to the model as the measurement log must describe a platform state in which a TPM Quote could have happened. Other events of interest could be exception handling: what happens when something unexpected occurs on the platform? What gets extended into PCR logs? Timing information might also be useful. These refinements all depend on the accuracy of the model for PCR extend actions and are left as future work.

One drawback of the proposed approach is that any access controls implemented by a higher level program (such as the operating system) might prevent an application from extending a PCR. This would have uncertain consequences depending on the error-handling code in place at the application. Furthermore, memory protection and isolation are not modelled at all which may result in invalid trust assumptions. Introducing access controls and memory protection would make the models significantly more complicated. However, it may sometimes be necessary. In the CSP IMA model in Figure 6.5, more filtering of messages on the 'extendreq' channel would be one approach to dealing with applications that are not given access to PCRs. Rather than replying with a 'finishextend' message, either no response could be given (resulting in deadlock for the application) or an error-handling routine could be specified.

The proposed approach could also be extended to non-PCR, software-based event measurement. For example, some virtual TPMs [14] use a trusted software component to record hash values instead of the TPM for performance and management reasons, as does the approach taken by Cabuk et al. [29]. These alternative measurement approaches still have the same fundamental abstraction and can be seen as just another set of PCRs. They could be modelled by this system without significant modification, although the trustworthiness of the 'soft PCRS' could not be assumed.

6.6.4 Late launch

The models designed in Section 6.3.1 and 6.4.1 assume a static root of trust, from the BIOS onwards. However, a dynamic root of trust would reduce the measurement list. Producing a model of a dynamic root of trust would be interesting future work, particularly as it involves a resettable PCR.

Indeed, this method would be immediately applicable for describing and assuring systems developed using the Flicker [131] and TrustVisor [129] systems described in Section 3.2.14. The sequence of executing PALs would be described on measurement logs, which would become increasingly complex, particularly if many are operating concurrently.

6.6.5 Multiple platforms

Assuming a complete platform state can be modelled using methods discussed in this chapter, the next step would be to combine PCR measurements from multiple platforms, to see if they would work together in a compatible and trustworthy manner. Particularly for web services, this would mean that composite services could be modelled usefully.

6.6.6 The TCG Platform Trust Service

An alternative to event reporting is using a trusted agent on the attesting platform to report on platform state at runtime. The TCG Platform Trust Service [208] is an example of this approach. The PTS software is measured as part of the trusted computing base of the platform and can then be requested to report on the integrity of important files and applications in memory. This approach has many advantages, providing dynamic, runtime information about the platform and supposedly avoiding the need to adapt user-level programs to support integrity measurement. It can also be used to report the content of configuration files and system settings.

However, the PTS has several problems. Firstly, it places a large and complex application in the trusted computing base of the platform. If the PTS is capable of analysing programs in memory, reading arbitrary files, and communicating with a third party, this presents a security concern. The OpenPTS project [155], for example, is made up of 16 thousand lines of Java (line count generated using David A. Wheeler's 'SLOCcount') and only supports basic reporting. Runtime compromise of the PTS would result in the platform becoming completely untrustable. While there may be no way to completely avoid this threat, a more robust and auditable approach would be beneficial.

Another problem is that the PTS is responsible for selecting the hardware and software that is reported beyond the trusted computing base [208]. This means that the attested state of the platform will rely on the challenger asking the PTS the right questions, and assuming it answers them correctly. While it might sound reasonable for the challenger to ask the PTS to measure files for it, it is not always obvious which files are important. Many applications can be configured using different files sometimes with a priority order. If the requester looks at the standard location for Apache's `httpd.conf`, for example, they may miss the configuration passed in at the command line. This becomes even more complicated when considering applications such as the JVM, where class loading rules are elaborate.

Furthermore, measuring configuration files is only one part of the problem. The challenger must understand how to analyse them to assess trustworthiness. In some cases this might be checking for one particular setting (PHP's `register_globals` being a good example of a single configuration setting with known security issues) but often it is more complicated. The general problem of being able to process and understand configuration files in any format appears to be a significant challenge. This is also true because the line between a static configuration file and an application is blurred. Some programs (such as Linux distributions) use bash scripts to alter the platform, and Java classes could also be considered data rather than code. Moreover, some

files contain sensitive information, such as passwords or port numbers, which a challenged platform will be unwilling to release. A generalised scheme for representing these files seems necessary.

The root issue lies in the fact that a PTS breaks application encapsulation. Only a program itself can know which configuration it is loading, and how it should be properly interpreted. The same is true for user input, environment variables and command line options. A PTS can be configured to attest some of this information, but full comprehension is probably infeasible. The burden of integrity measurement should fall on the application, not an external runtime agent. Furthermore, every PTS configuration will be entirely application and context specific, with no *generality*. This means that companies will be forced to independently produce policies for similar situations.

6.7 Comparison With Related Work

Some of the techniques used in this chapter refer to fundamental computer science concepts. Applications are modelled similarly to concurrent hierarchical state charts [120], with the Prolog model allowing similar transitions and structures to Petri nets. Some of the models shown in this chapter are sequential but non-deterministic, and can be considered Kripke structures [27]. However, having multiple concurrent Kripke structures means that process algebras are appropriate for modelling each component. A great deal of literature exists on CSP [174] and its use for model checking systems, including the FDR tool and trace refinement [64].

In trusted computing literature, process algebra and model checking have rarely been used. Rohrmair [173] has analysed trusted computing protocols in CSP and created similar boot models to those presented in this chapter. However, the focus of his thesis is *verifying* the integrity reporting process to identify attacks, rather than aiding the challenger in identifying trustworthy platforms. He demonstrates that simple time-of-check-time-of-use issues are present, and suggests that a trusted agent might help. He also identifies that the measurement lists may not scale and might become unmanageable. The main difference with the work presented in this thesis is that CSP is used as a method for identifying platform state, and is not used as a formal verification tool in the same manner. Indeed, it is just one step in a more complex process. Furthermore, this chapter includes many more examples and an alternative implementation. In other related work, Pitcher and Riely [165] use a form of the π-calculus to specify and check enforcement of access controls on attesting platforms.

The event reporting approach has been used before, and perhaps the most relevant work is in various papers by Naumann et al. [147] and Alam et al. [4] on Model-based Behavioural Attestation. They propose to log and attest to all behavioural updates. Their verification framework has been refined for enforcing usage control, rather than the validation of reported attestations. The requirement for a domain-specific verification process is common: the approach suggested by Naumann et al. [148] for the Android platform requires a new process, as does the UCLinux [111] platform. These example strengthen the argument that a *common* and *composable* method for describing PCR event-measurement schemes is required.

Log verification with trusted computing technology has been discussed by Huh and Martin [93], although with a focus on maintaining the integrity of logging components, rather than verifying the meaning of the logs themselves. Semantic Remote Attestation (see Section 3.2.4) tries to make attestation more meaningful, but does so by attesting runtime properties as logged by a virtual machine. This approach is reasonable, but requires all programs to be running under the trusted VM. It would be possible to model semantic attestation within the framework proposed in this chapter.

Another approach for verifying integrity reports (and integrity measurement) is taken by Datta et al. [53]. They use a concurrent programming language to specify, in detail, exactly the operations performed during authenticated boot, in order to prove properties such as code execution. Their approach is low-level, focusing on trusted computing primitives, memory separation and protocols. Their concern is primarily with proving safety and security properties, rather than associating high-level system state with attestations. Whether their approach will scale to a real runtime system is unclear. Conversely, the approach taken in this chapter is intentionally high-level, and assumes properties such as memory isolation and the trustworthiness of PrivacyCAs, and software implementations. Abadi and Wobber have also attempted to reason about attestation using an authorization logic [1].

A further solution to the increasingly complex task of integrity measurement in a *virtualized* platform is given by Cabuk et al. [29]. They propose using standard integrity measurement to measure the usual boot process up to a new, pre-VM component called the 'Software-based Root of Trust for Measurement (SRTM).' The SRTM is responsible for collecting further measurements from VM instances. This provides a hierarchy of measurement, and is further elaborated to provide a solution to different integrity dependency models, including completely independent virtual machines and those that depend on each other or common shared components. The relation to this chapter is that *integrity dependence* models ought to be mirrored by *integrity measurement* models. The trustworthiness of the integrity measurement system is dependent on whether this is the case. Interesting future work would be to integrate these concepts further, or perhaps derive the true dependence model from the measurements provided and identify any discrepancies.

6.8 Conclusion

Existing approaches for interpreting measurement logs are generally restricted to only supporting a simple whitelisting policy. More sophisticated assurance requirements are difficult to implement, and generally either rely on a runtime measurement agent [208] or are ad-hoc and specific to a particular problem [148]. This chapter provides an alternative, a framework for any system which uses integrity measurement to report on more detailed behaviour of the platform. This allows a picture of platform state to be built from the measurement log in combination with models of program behaviour. Two implementations have been investigated, using CSP and Prolog. Issues with both have been discussed and act as a good starting point for future development of a generalised, standard verification system. However, more

work is required in order to overcome some of the issues, particularly the description memory protection and dynamic roots of trust. From the experiences gained by modelling several real components it appears that a combination of the approaches used in this chapter will be appropriate.

The strength of this process – the fact that it does not require a custom process or runtime agent – is also a limitation. Applications must be accurately modelled and not suffer from unrecognisable remote attacks. While the accuracy of the modelling is probably reasonable, as PCR-usage models are small and simple, this is still a concern. Potential solutions to this problem have been discussed in Section 6.6.2 and a further analysis is carried out in the evaluation. This is left as future work.

However, a more important limitation is that models only describe PCR usage, not overall behaviour. As a result, users still cannot be sure of the behaviour of a remote platform. This is particularly true if they have no experience of using it in the past, or if it is running custom-built software. This is true of most web services: they offer unique functionality, and do not have readily available source code for analysis. The next chapter looks at connecting more detailed functional behaviour to attestations, with exactly this problem in mind.

Chapter 7

Uniting Program Definition and Platform Attestation

Remote attestation suffers from the *semantic gap* problem, explained in Section 3.1, since integrity measurements describe only the *execution state* of the platform, not its trustworthiness. While the previous chapter demonstrated how to link integrity measurement to some notion of platform state, it is still not easy to work out whether the platform will behave as expected. The argument put forward by Proudler [169] and the Trusted Computing Group is that this knowledge can be gained through *previous experience* of the platform and applications in question. As discussed in Section 2.1.1, however, this may not be sufficient for web services. In particular, remote services are hard to gain past experience with, as their implementations may be updated frequently.

In addition, web services are likely to offer some form of unique functionality, which must by definition be unknown to external users. Attesting to custom-built software [134] is difficult because of the lack of reference values to compare against. No public hash values will exist of the application, except those published by the developers themselves. Should the developers introduce flaws, maliciously or accidentally, then this reference measurement has little purpose. Service users can only hope that the service was implemented correctly.

To bridge the gap between attested binaries and platform behaviour, this chapter looks at creating *compile-time guarantees* of service applications. A shorter version of this work was originally published in conference proceedings [121].

7.1 Attesting Platform Behaviour, Not Execution State

Rather than attesting to the binary image of an application, it would be more useful to know the source code that built it. Although there are dangers in assuming that the semantics of the source code are directly implemented by the binary [217] it can be argued that the increased behavioural knowledge obtained from the source code is still a significant advantage. Furthermore, the area of code analysis has been well-explored, with known techniques for

proving properties of small applications (see Section 2.4). If it were possible to attest to software with *proven* code properties, the *semantic gap* problem would be reduced.

A basic scheme, therefore, would be to release the source code of the application that needs to be attested and let users analyse it themselves. They would then know what it can do, and be able to compile it, creating a comparison hash value. When the application was run by a remote platform, and attested, they would be able to match the hash value present in PCRs against their locally compiled version. There are a number of assumptions necessary for this to work in a web services scenario:

- The user must be capable of analysing the code. This may depend on the availability of numerous code libraries, operating system features, and so on.
- The middleware and OS running at the web service must also be trusted by the user.
- All important configuration settings must be made available.
- The middleware and OS of the web service must be *attestable*. Each part of the software stack must support integrity measurement. It must be possible for the user to receive a remote attestation and interpret it. This implies the existence of an integrity management infrastructure, which has a whitelist of trustworthy pieces of software. Every binary running on the service platform will need to be on this list.

While the last three points have been addressed in previous chapters, a number of practical problems remain. Application providers may be unwilling to release their source code, particularly if it offers some form of unique functionality. Users may find it difficult to recreate the same build environment as the provider, which would alter the comparison hash value. Much more significantly, the code analysis may be difficult, and it is unreasonable to expect all end users to do it themselves. One solution might be to devolve compilation and verification responsibility to a trusted third party. However, the introduction of an additional party would be worth avoiding if possible, as attestation already suffers from too many trusted authorities (see Section 3.1).

The rest of this chapter proposes to use remote attestation to allow the *provider* to perform the verification process and then demonstrate that they have done so to the user.

7.2 Trustable Remote Verification: Establishing Properties Without Source Code

There are two main approaches to program verification. It can be done by a trusted third party, but they may charge a high price for their services. The alternative is to verify an application locally: if source code can be inspected before compilation, any errors can potentially be spotted before the application is run. However, given the size of any complex application, even a highly skilled programmer would struggle to spot potentially erroneous behaviour in source code. This has been improved by Proof Carrying Code [149], where the majority of the

effort is carried out by the application distributor. However, this is not a suitable solution for web services, where all applications are running remotely. Users have no idea what source code is being run at the service, and have no way of verifying it. Neither third-party nor local analysis can therefore be considered appropriate for service-oriented computing.

The rest of this chapter introduces the novel concept of *trustable remote verification*, a way to let the *provider* perform verification and then prove to users that they have done so. An overview of how this is implemented using trusted computing and the authenticated-boot process is shown in Figure 7.1.

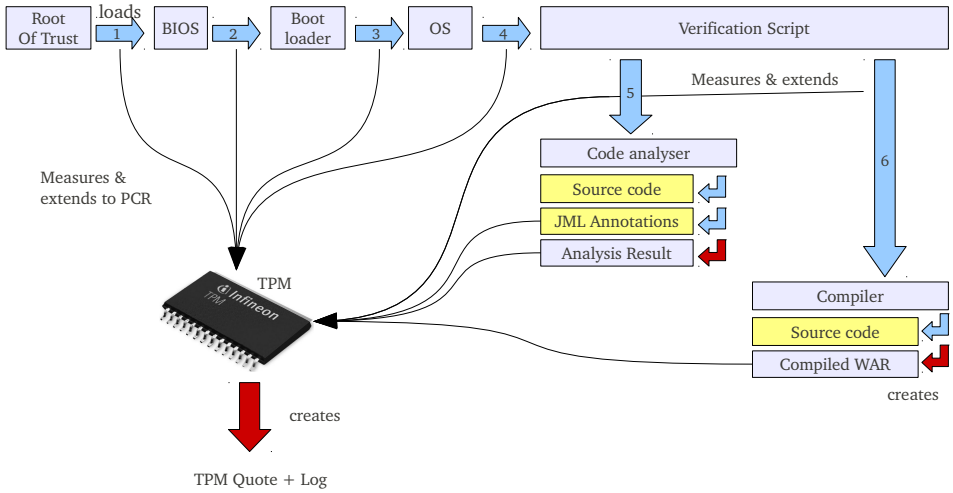

Figure 7.1: An overview of the trustable remote verification process, showing the order of execution and all items measured into PCRs.

7.2.1 Overview

The principle behind trustable remote verification (TRV) is the use of TPM attestations as long-term credentials. The service provider (W) performs program analysis using a local machine (the verification platform, V) and then attests the result. The relying party can then check the attestation to see if the verification process has been carried out properly, and what the results were. To do this, V must authenticated-boot into a trustworthy OS, which measures and extends each step into a PCR. After boot, the annotations (W_{ann}) which represent the service contract are measured. These will specify some important property of the service which the requester requires (see Figure 7.2 as an example). Then a program verifier (TV) is measured and loaded, and the source code (W_{src}) is analysed against its annotations. The result of this step (TV_{res}) is also measured and extended into a PCR. Next, the source code is compiled by a trusted compiler, TC. A hash of it and all the compiled binaries (W_{bin}) are measured and extended. At the end of the process, a quote is produced which contains two PCRs, holding measurements:

```
/*@ requires
  @   accFrom != null && accTo != null && amount > 0;
  @
  @ ensures
  @   ((accFrom.getBalance() == \old(accFrom.getBalance())) &&
  @      (accTo.getBalance() == \old(accTo.getBalance())) ) &&
  @      (errLog.content.theSize == \old(errLog.content.theSize+1))
  @   ||
  @   ((accFrom.getBalance() == (\old(accFrom.getBalance()) - amount)) &&
  @      (accTo.getBalance() == \old(accTo.getBalance()) + amount) &&
  @      (transLog.content.theSize == \old(transLog.content.theSize+1)));
  @*/

public void makeTransfer(Account accFrom, Account accTo, int amount) {
      ...
}
```

Two outcomes are specified in the above code: either the account balances change in the expected way, or both remain the same. An entry is added to the transaction log in the first case, and to the error log in the second.

Figure 7.2: An example web method, complete with JML annotations.

$$W_{quote} = Quote_{AIK-SK(W)_1} \left\{ \left\{ \begin{array}{ll} pcr_{0-10} & = \{ \text{ boot process } \} \\ pcr_{11} & = \{TV, TC, W_{ann}, TV_{res}, W_{bin}\} \end{array} \right\}, nonce \right\} \tag{7.1}$$

This is a credential, which will be used by the provider to show that a program binary, W_{bin}, was compiled from source code which was verified against its annotations, with analysis result TV_{res}. In the ideal case, TV_{res} would state something simple such as 'verified.' The credential can be checked by making sure that TV, TC and the boot process are all trustworthy, checking that TV_{res} does not show any errors and finally verifying that the annotations are sufficiently strong for the program to be trusted. Note that a nonce is unnecessary as the freshness of this credential does not affect its trustworthiness.

7.2.2 Assumptions

Trustable remote verification relies on several assumptions.

- The platform performing verification has a valid TPM which has not been tampered with.
- There exists a *verifier*, a piece of software which can read the program contract and source code and automatically decide whether the latter corresponds with the former. This must be trusted to work properly by the client. In the proof-of-concept implementation, JML annotations are used as a contract and ESC/Java2 is used for verification.

- There is a simple operating system, again trusted by the client, which the verifier can run on without interference. This OS can measure every step of its boot process into PCRs.
- The verifier, compiler and operating system have SHA-1 identities known to the client.
- Any third-party libraries that the verified application uses are either annotated and verified with the service, or their identities are published by the server and trusted by the client.
- All configuration files used by the web service or the verifier are made available to the client.

7.2.3 The new chain of trust

TRV decouples the process of certification and application execution. Once the web service binary has been verified, it can be run on any service which supports secure boot, and the same credential can certify it. This is desirable from an end-user perspective, as the amount of effort required to establish trust in a set of remote services (perhaps implemented for load-balancing reasons) is greatly reduced. The disadvantage is that the chain of trust is now longer, and contains potentially two TPMs, one for the web server (TPM_W) and one for the verifier (TPM_V).

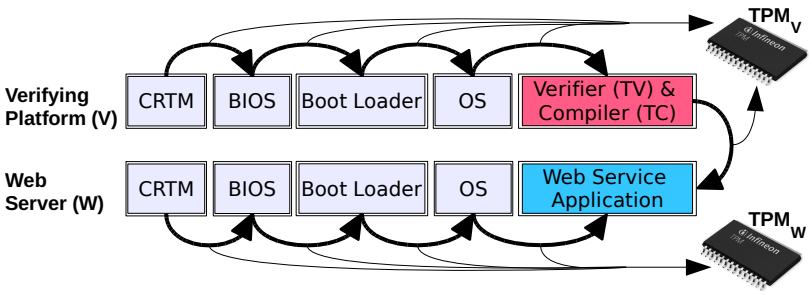

Figure 7.3: The chain of trust for trustable remote verification, showing execution order and measurement storage.

7.3 Prototype Implementation

Two parts of this system were implemented: the credential-creation stage on V and requester validation stage at R. In the prototype system, services are written in Java, with methods from one class exposed as a web service. This class is annotated with JML assertions, which are the properties that this service promises to fulfil. ESC/Java2 is used as the program verifier (TV), and Ant plus the standard Sun JDK are used as the compiler (TC). The result of compilation is a WAR file which is run from the Glassfish Application Server.

7.3.1 Server credential creation stage

The program verification stage requires the following steps:

1. The service source code and configuration files are placed onto V.
2. An AIK certificate is obtained from a Privacy CA.
3. The trustworthy OS with secure boot is started.
4. The OS measures the JVM, and then runs the verifier. This measures the following items into the TPM:

   The front-end JML annotations of the service.

   All libraries and files necessary for compilation.

   The WAR archive (binary) created by compiling the service.

   The output of running ESC/Java2.
5. A quote is created, signed by the AIK, containing two PCR values. One has the current OS and application measurements and the other has all the measurements made by the verifier.

Additionally, an archive (see Equation 7.2) is created to help the service requester validate the measurements. This includes references to external libraries, ESC/Java2 output, JML annotations and a log of the entire process. It is used by the requester to validate the process later on. References to external libraries should point to where the end user can download the library to verify its identity.

$$W_{arch} = \{W_{quote}, [\text{ libraries }], W_{ann}, TV_{res}, \text{ log }, [\text{ config files }]\} \tag{7.2}$$

7.3.2 Credential validation steps

The service requester, R, must obtain and verify the credential that was created using the steps in Section 7.3.1. This requires the requester to download W_{arch} and W_{quote} and then do the following:

1. Check that the software running on W is trustworthy, and that the web service application has the identity W_{bin}, equal to the one in W_{quote}. In other words, make sure that the executing web service binary matches the compilation output of the server credential creation stage.
2. Check the AIK used to sign W_{quote}.
3. Check that V's OS and boot process are trustworthy.
4. Check that the verification program (including TV and TC) is trustworthy. This would probably involve checking against a public list of verifiers (with available source code) which are known to be sound and complete.

5. W_{arch} contains a log of the verification process, which will need to be checked against the PCR values held in W_{quote}.

6. Each individual step described in the log must now be verified. The server must provide any comparison resources that the client does not have access to. This includes:

   All configuration files used in the verification process.

   The external libraries used and any assumptions made about them.

   The verification result itself.

   The verified service annotations.

Additionally, some useful service properties must be described in the annotations, and the verification result should not show any situation where they do not hold. In the prototype implementation, some of the checks described are performed manually, but it would be feasible to create automated tools. Part 1 and 3 require an integrity management infrastructure, such as IMI [142].

7.3.3 Configuration files and compilation with Ant

Because the credential-creation step is complex, involving program compilation and creation of web service artefacts, there are a number of configuration options. These could potentially make the verification process untrustworthy (for example, running ESC/Java2 on one piece of software and then compiling and measuring another). Therefore, the configuration files are measured and included in W_{arch}.

Program compilation can be complicated, involving libraries, configuration, and archive creation. As a result, most Java developers use Ant rather than just `javac`. For the compilation step of the prototype, the same issues are present, so the Ant build file approach seems sensible to reuse. However, Ant is an extremely powerful program and it might be possible to write a malicious build file. This could make it appear that the program was compiled verified when it actually hadn't been. In order to stop this from happening, the build file must be measured into the quote and included in W_{arch}. It must also be checked by the requester, along with all the libraries and files it references. In the prototype this must be done manually. This problem could be avoided (to some extent) by insisting on the use of a safe subset of Ant, rather than allowing the whole set of features.

7.4 Evaluation and Observations

7.4.1 Benefits of trustable remote verification

In trustable remote verification, it is possible to determine, with a fairly high level of assurance, *something* about what a remote service will do when invoked. This something will range from a complete logical description of the functionality of the service, down to perhaps a simple invariant. In the example given in Figure 7.2, the new assurance property is that the method

`makeTransfer` will at least revert back to our previous state rather than fail in an unknown way. The error log is also guaranteed to keep track of any failures. In terms of security, assertions about information flow could be used to be sure of confidentiality. JML has also been used before for security properties [158]. Importantly, the remote verification process is independent of the verification and specification tool, so any can be used.

No additional third party is required to create the service credentials, beyond a Privacy CA which is likely to exist already. The client and server-side code needed to implement these features is fairly small (the prototype is under 3000 lines of code), with the only significant extra requirement being an operating system that supports secure boot. Furthermore, this system allows for software update, as each new version of a service can be re-verified and a new credential produced. This can be part of the standard build-cycle for a project. Another key benefit of this system over the basic architecture is that the source code of the service never needs to be revealed, not even to a third party. This would be attractive to a company with valuable or confidential code.

7.4.2 Trustworthiness of the architecture

The strength of TRV can be measured by how difficult it is for a provider to *falsely* claim that a service has certain properties. Any system can be broken through weaknesses in its trusted components. In TRV, these include one (or more) TPMs, a verification OS, verification tool, compiler and the software stack running at the web service. TPMs are designed to be immune from software attack, and hardware attacks are non-trivial. They are therefore unlikely to be the weakest point in the system.

The verification environment needs to be a trusted component. However, the verification OS can be small and simple and only needs to be able to run a program verifier and compiler. It does not need network access, or the ability to accept user input at runtime. Future CPUs which run bytecode might be a good way of avoiding vulnerabilities, as might a microkernel-based OS. The verifier and compiler, on the other hand, are a bigger issue. They are necessarily complex systems, which accept input in the form of program code and configuration files. Arguably, however, compilers must already be trusted, and there are several open source compilers which have gone through considerable scrutiny. A weakness in the verifier would be a problem. If it produced false negatives, it could then potentially certify a system which does not maintain its properties. There is no obvious solution to this problem, but creating one acceptable verifier or compiler is likely to be easier than creating many perfect applications.

Perhaps the most significant trusted element is the rest of the software running at the web service. If a bug or vulnerability causes it to behave in an unexpected manner, then the properties guaranteed by the web service application are irrelevant. This is why, in Chapter 5, an architecture for limiting service middleware was proposed.

Overall, TRV is limited in the level of trust it can establish, and is not appropriate for extremely high assurance systems. Instead, it would work best as an additional check for service providers who are attempting to improve their perceived reliability in the marketplace. In such a scenario, one threat is that a company with normally good intentions tries to subvert

the system for a new version of their service. They might try to rush a new feature, at the expense of verification. TRV would make this much more difficult to do, and so the provider would be more likely to spend the extra effort in verification. However, the strength of the guarantee is directly linked to the verification tool, so higher assurance might be possible in the future.

7.4.3 Multiple verifications

One useful property of this system is that the credential-creation process is entirely separate from the runtime attestation. As a result, the prototype can be extended to offer multiple, potentially independent verifications and certificates of the same service. For example, one service provider could first verify their source code with ESC/Java2, producing a certificate, and then do the same with an alternative program analyser. This might satisfy users who will only trust a particular analysis program.

Furthermore, multiple organisations can verify the same service. Assuming they are given the source code, they can all independently run a verifier and produce a certificate. This significantly strengthens the chain of trust, as it is no longer 'anchored' by just one TPM. The problem highlighted in Figure 7.3 – that two TPMs are now trusted – is no longer as significant a problem, as the verifier chain can be repeated on different platforms, each one increasing the trustworthiness of the chain itself. This might be useful for high-assurance systems, such as e-voting.

7.4.4 Verifying multiple services

Verifying services which themselves contact *other* services have not been considered. This is a common scenario, and a significant limitation of the prototype. However, there do not seem to be any obvious reasons why any services which have also followed this scheme could not be incorporated. These 'sub services' could be wrapped by a stub object, which asserts the same annotated properties. This would not be verified, and instead all the certificates could be presented to the user. Implementing this in a user friendly and secure manner would be a challenge. There would also be other difficult problems, such as what to do when one of the sub services is no longer considered trustworthy.

7.4.5 Attestation as proof of execution

It is possible to generalise trustable remote verification (and similar work) to use TPM attestation as a mechanism for generating proofs (in the informal sense) of program execution. TRV is one example, as in fact the attestation just proves that a compiler and static checker have been executed and produced a certain result. There are numerous other possibilities, as any program can be shown to have run and produced a result. However, it is not possible to show that a program has *not* been run, and full program verification is required to show that the result is in any way useful or *correct*. However, this is still a useful feature. A general model, using only TPM attestation, can show this technique fully.

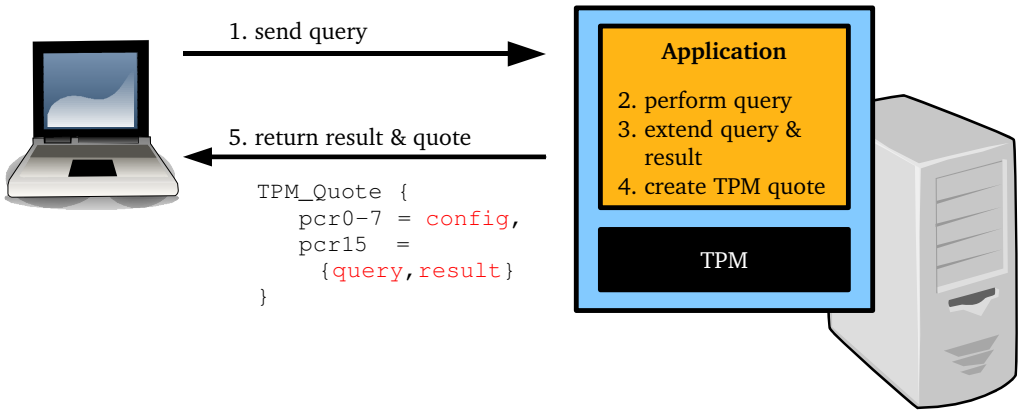

Figure 7.4: Using TPM attestation to produce proof of execution

One issue is that the values extended for one proof of execution must then be included in the attestation for every subsequent request. This can be avoided by using a resettable PCR for the result of the computation, although this has different security properties. As the trustworthiness of the software running on the platform must be relied upon anyway, using a resettable PCR should not alter the trustworthiness of the resultant value. However, this might not be the case if the machine was compromised at runtime, the PCR reset, and a new value inserted. Use of a resettable PCR is therefore only sensible when the software is resistant to attack. This point is important, as the main benefit of providing a TPM-based proof of execution is that it cannot be forged, even by an insider.

Optionally, a time stamp could be included in the query. This would be useful if the computation was time-sensitive. Similarly, more contextual information, such as the platform runtime state, could be extended to PCRs if it affects the outcome of the query.

This general approach is appropriate in many situations. For electronic voting, it might provide proof of a cast vote (see Section 8.2.6). For a remote data source, it could provide provenance information for the result of a query. In each case, should a vulnerability or bug be discovered in one of the programs included in the measurement log, it would be possible to trace which results were affected. If, for example, a badly implemented algorithm produced inaccurate results, its implications could be discovered after the fact. This could be useful in e-science and grid computing scenarios.

The advantage of tying execution to a TPM-generated guarantee is that it becomes difficult to *maliciously alter* a result. This would not be the case if the result was signed by a key held in software. Furthermore, this could combine information about the AIK to guarantee which machine generated the result. See Appendix C for more details.

7.5 Alternative Implementations and Approaches

An advantage of trustable remote verification is that no specific compiler, source language and verification technique is required. The most appropriate one can be used in any situation. There are several possibilities beyond JML and ESC/Java, some of which are discussed in this section.

7.5.1 Trustable remote *compilation*

Is there anything to be gained from removing the verification step and simply attesting to the compilation process? There are situations where this may be useful. For example, parallel work by Meng et al. [134] suggests that this would be useful for showing how a standard application has been modified. In addition, it could be used to identify which bits of the system need to be attested: those which affect the properties of the system. Alternatively, this could be used to demonstrate, for example, which patches have been applied to a standard Linux kernel. More exotic properties require a more sophisticated compiler, several of which are discussed in this section.

Another advantage of attested compilation is the added provenance information. While most developers will know which compiler they use each day, it is unlikely that many know which compiler was used to create the related libraries or the compilers themselves. If an earlier compiler was shown to be weak, or contain a hidden flaw, its full impact could be found by referring to the chain of compilation certificates. Assuming the original compiler was not flawed, any step in the compilation of subsequent compilers that is known to be untrustworthy could be identified. This would make Ken Thompson's "Reflections of Trusting Trust" [217] attack more noticeable.

7.5.2 Booster

Booster (see Section 2.4.4) is one way of implementing trustable remote verification. The compilation phase must include the Booster code generator, and the verification process need only interpret the formally-described model in the build file. This relies on the fact that the Booster code generator is correct, and will produce an end program with the same properties as its specification.

The user could be given the entire build file as a property to rely on. This is similar to releasing full source code, but would be easier to analyse. Alternatively, *properties* could be derived from the specification in the verification step. These properties can be potentially fine-grained, such as 'no user can create a report and sign it off' and about model correctness: 'all students have a supervisor and all supervisors are member of a department.' Some work has already been done on reasoning about Booster specifications, including properties relating to intra-object behaviour [31]. The certificate issued by the verification platform would contain the list of properties guaranteed rather than the entire model.

Using properties rather than a complete specification makes it much easier to deal with

software upgrades. The issue is that if the model needs to change, perhaps to accommodate a new data field, it must be re-compiled and lose all data sealed to the old application measurements. Instead, any model with the same properties should maintain access to the old data, no matter how many small changes are made. One solution is to use a 'sealed-key' approach to encrypting data (see Section 3.2.12), much like in Chapter 5. When the model is compiled, the certificate created by the trusted party includes a public key. Users encrypt sensitive data with this key when it is sent to the service. The private part of the key is issued to the service provider, but sealed to its current software measurements. It is therefore accessible only when this specific compiled version is running. However, the difference comes when the model is re-compiled. This time, the same private key can be issued to the service, sealed to the *new* measurements. Old data therefore remains accessible. The only time when the private key would not be re-issued is when the newly compiled model no longer fulfils the same properties that the original did.

The advantage of this modification is that small changes to the model are no longer a problem for the user or provider, as the same keys are used. The service provider can keep using old data. However, it does mean that providers must be careful with the properties they decide to guarantee. If too many are specified at the beginning, failing to meet one later will result in a loss of stored information. It should be noted that the validation of other software components will also need to be modified similarly to allow for upgrades.

In addition to specifying Booster properties, the model-driven approach may have other advantages. In the previous chapter, one of the disadvantages to building a system model was that the model itself had to reflect real application behaviour. The only thing linking the two was the signed word of the application developer. With trustable remote verification, however, a model-driven build process (demonstrated by Booster) could be reported, demonstrating model compliance. This means that a previously unknown application can be fit into the modelling scheme without difficulty.

7.5.3 Runtime checking

A similar but alternative approach to the one taken in the prototype is to use the JML compiler [112], `jmlc`, as opposed to the Java compiler. `jmlc` adds runtime assertion checks into the program bytecode in addition to normal compilation. These will raise an exception if any of the preconditions, postconditions or invariants fail. Except for a small performance hit, this behaviour is transparent unless one of the assertions is violated.

The advantage of this scheme is that static analysis is no longer necessary. As a result, runtime components such as configuration files and databases can be accessed without breaking the assertions. However, the downside is that an untrustworthy service can make promises, and then break when they are not fulfilled. This might result in lost data and an unreliable system. The best this would be able to say is that *if* the service does not fail, it will work as expected.

7.5.4 Runtime input validation

A specialised use of the runtime checking approach is to do runtime input validation. When a web service receives data, it is first checked against the XSLT schema to make sure it is syntactically sound. However, this does not show that the structure is *semantically* correct, and could be maliciously (or unintentionally) formed to disrupt the service. To avoid this, JML pre-conditions can be placed on the inputs of the web service front-end. Any messages failing to meet these conditions will automatically be rejected. By specifying this behaviour at compile time, users can find out the exact conditions on their requests, and be sure that malicious input will not affect the service at this level. If it is assumed that XML-level vulnerabilities have been eliminated with the split-service architecture described in Chapter 5, then this provides further defence in depth. The JVM itself could have vulnerabilities, however, which are not mitigated.

7.5.5 Using a PAL-based compiler

Another implementation option would be to take advantage of the Flicker [131] and TrustVisor [129] systems described in Section 3.2.14. Assuming compilation and verification could be performed in the limited late-launch environment, this would reduce the size of the chain-of-trust on the verification and compilation platform significantly.

7.6 Comparison with Related Work

Several pieces of existing work cover broadly similar approaches to trustable remote verification. Both the SAConf system [232] and work by Meng et al. [134] work in the same way, and were developed in parallel with this (and published after [121]). SAConf is proposed to be used for validating configurations and policies, and is not separated into the certification and attestation stages as proposed in this chapter. The Tisa system proposed by Rajan and Hosamani [170] has the same goal as TRV, but is implemented through a runtime monitor attached to the web service. The advantages are that the service itself does not require modification on annotation. However, the runtime monitor must be configured correctly, and it must be possible to gain meaningful information from the traces and execution history recorded. Moreover, it only provides a history of actions, rather than a statement of future trustworthiness.

Certified compilation has also been the topic of much prior research. Hornof and Jim [91] describe a system which provides a type-safety certificate for a subset of C. However, it is assumed that the verifying party will perform the compilation themselves, rather than attestation of this process. Appel et al. [8] have built a trustworthy proof-checker, designed for Proof Carrying Code systems. This is particularly promising, as they have a TCB of only 2700 lines of code. This would minimize the chance of an attack on the verification platform. They also have a good discussion of the difficulty of trusting compilers and verification.

7.7 Conclusion

This chapter has proposed a new mechanism for assessing applications without access to their source code. This has the potential to reduce the *semantic gap* problem of attestation for service platforms. The trustable remote verification approach is independent to the software verification mechanism used, can be strengthened with multiple verification stages, and certificates can be reused on different platforms. In combination with a minimal service infrastructure, and trusted compilers and verifiers, it can finally allow a service provider to attest to their current execution state in a *meaningful* manner. When combined with the modelling approach described in the previous chapter, previously unknown applications can be composed together with the rest of the platform to give a coherent and attestable picture of the entire system.

The last three chapters have introduced individual parts of this solution. In the evaluation, the complete set of improvements are assessed to see how much more feasible remote attestation of web services has become.

Chapter 8

Evaluation

The last three chapters have introduced solutions to the practical problems associated with attestation and integrity measurement. This chapter will evaluate these contributions to determine to what extent attestation can now be considered a feasible technique for assurance in service-oriented computing. This is done by implementing an example service using the methods proposed in the last three chapters. After explaining the evaluation approach in Section 8.1, the example service is described in Section 8.2. Sections 8.3 and 8.4 provide an analysis of the implementation and Section 8.5 identifies the attestable assurance properties. Finally the thesis question is reconsidered in Section 8.6.

8.1 Evaluation Approach

To evaluate the contributions made in this dissertation the original problems with attestation, as described in Chapter 3, must be revisited. These included *privacy*, *semantic gap*, *runtime attacks*, *whitelisting*, *trusted parties*, *performance*, application *compatibility*, *trusted path* and *multiple domains*. However, methods proposed to tackle these issues, as described in the previous chapters, first need to be combined together to present a realistic platform for evaluation. Although a few components were not implemented, most were, and the next section presents the resulting system.

The modified system will then be assessed with regard to how much assurance can be gained from attestations. The focus will be on practicality and security, as well as looking at other associated overheads. For example, the use of trustable remote verification certainly helps to provide additional assurance, bringing attestation and behaviour closer together, but does require more effort from the developers and relying parties. It also has some potential vulnerabilities. The key metrics will be improvements to the issues discussed in Chapter 3.

A ballot box for e-voting was chosen as the evaluation scenario. This is because it has a limited number of features, making it possible to implement easily, but has security requirements. Particularly, it must maintain high integrity, despite being a reasonable target for attack. Furthermore, it can be designed to take advantage of TPM features (making attestation

a sensible approach), and may rely on communication with another party, making the problem more interesting. It is a similar problem to that of Attested Append-only Memory (see Chapter 3.2.3), often discussed in related literature.

8.2 The Complete *Attestable* Service Architecture

Appendix A presents the specification of a ballot box service, designed to provide voting integrity. The TPM's monotonic counter and PCRs are used to count the number of votes submitted, and attestations provide evidence of successful ballot submission. This is a novel contribution in itself, as well as using principles described in the literature on Attestable Append-only Memory [43, 113]. More interesting, however, is how the service has been implemented so that attestation does not suffer from the problems described in earlier chapters, particularly when using PCRs for integrity measurement *and* recording votes. The system has been implemented (apart from a few components) using the techniques described in the last three chapters. Details of the development approach, integrity measurements and verification procedure are outlined in the rest of this section.

8.2.1 Assurance goals

The goal of using trusted computing for a ballot box is to provide assurance in the integrity of the voting process. Informally, the aim is to prevent votes from being modified after they are collected. The following properties are guaranteed, assuming the software is trustworthy:

1. No message received will be modified. This property holds providing a secure channel is established between the voter and the ballot box, the ballot box has been implemented correctly, the correct software has been run (the authenticated boot-process pcr_{0-7} should demonstrate this), and that the ballot box has not been compromised at runtime.

2. If the ballot box can report a fresh attestation and integrity measurement log which lists every message and skips no counter values, then every message the application received was recorded.

3. If any counter values have been skipped, the implication is that messages have been lost. The period in which these messages were lost can be seen in the last complete timestamp. Nothing can be inferred about these messages, as they could be due to the platform being booted in another configuration and modifying the counter.

4. The order in which messages are recorded is the same as the order in which the application received them.

However, the platform cannot guarantee that no messages were lost due to potential hardware failure.

8.2.2 Architecture

Two services were required to implement the ballot box: the *ballot box (BB)* itself, where votes are submitted and collected, and a *time stamp (TS)* service which could be requested to regularly certify the current state of the platform. Both of these were split into trusted and untrusted components as described in Chapter 5. This was automated by the build scripts (see Section B.1) to make the process as simple as possible. The RMI interface, web service and WSDL are all generated during compilation. These were then deployed on different systems and connected together. For the evaluation, both back-end services were placed on one platform, supporting authenticated boot, and the front-end services on another that did not.

One complication was the communication between the two services. The ballot box service had to request a timestamp regularly. As the two services were both on the same platform, this could be done through simple RMI calls. However, in the general case this wouldn't work. The solution was to provide a proxy on the untrusted platform, that could forward requests from the ballot box back-end to the time stamp front-end, and back again. Figure 8.1 shows all the components. However, it is likely that the 'Time Stamp RMI Proxy' and 'Ballot Box Front-End' would be the same platform running two different processes. The proxy was not automatically generated, but it would be straight forward to do so.

The result of this architecture is that only the two back-end services need to be attested by a trusting party. As these are combined onto one platform, only this needs to be integrity-checked.

In terms of software configuration, the back-end server is running a similar set of programs to the platform used in Section 4.2. The same version of the kernel is in use, as well as the same integrity-measuring JVM. The front-end is running the Glassfish application server.

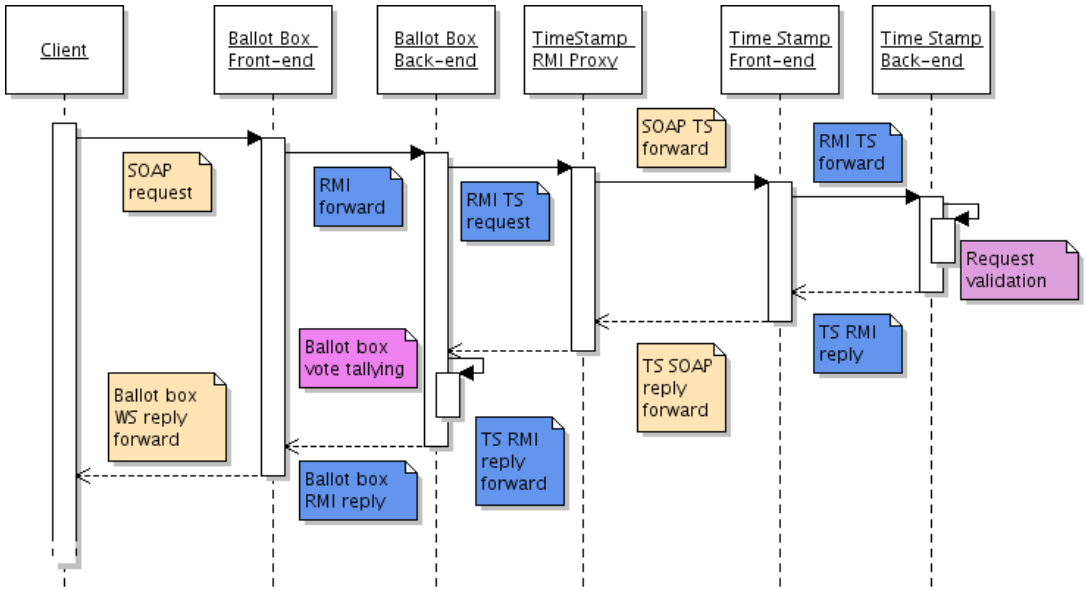

Figure 8.1: Sequence diagram of a request to the ballot box evaluation example

8.2.3 Writing and compiling the code

The services were written in Java, and annotated using JML pre- and post-conditions, in a similar manner to the approach used in Chapter 7. However, rather than using the ESC/Java2 checker, the JMLC tool compiled the code to automatically introduce condition checks. A simplified build script for the Time Stamp service can be found in Section B.1. This demonstrates the compilation of the base service code, then the JML-annotated interface, and then automatic conversion to an RMI and web service. This file also shows how the build process goes about measuring the compiled output, using TPM extend, TPM quote and a log file.

The code itself is unremarkable. The ballot box service interface has some pre- and post-conditions applied. These are shown in full in Figure 8.2 and include:

- The `getCurrentVotes()` method, which returns a list of ballots, was defined to be `pure`, having no side-effects.
- The `addVote(...)` method ensured that after being run, the `getCurrentVotes()` method would return a greater number of entries. This could also state that the last entry was the same as the argument given.
- The `getHistory(Date start, Date end)`, which returned all logs of ballots cast between a certain date, was guaranteed to return only contiguous records. It would also only return logs with a start date before the required `end` date, and an end date after the required `start` date. This requirement was implemented as a separate verification method.

No suitable conditions were applied to the 'checkpoint' method, which saved all current ballots to a file, as well as an attestation. This was because it is not a pure method (it does change internal state), but it has no visible side-effects to the user, except altering the result of future `getHistory(...)` requests. This is difficult to describe concisely. The Time Stamp service also had JML conditions, mostly for simple type checking, including a pre-condition that the ballot box sent a valid AIK credential.

8.2.4 Server administration

On the back-end server, the TPDMenu (see Section 6.5) shell was installed to allow administrators to log in, but be constrained in their actions. In this case, administrators could start and stop the RMI services, as well as the RMI registry. Root login into BASH is also possible, but should result in the invalidation of a PCR, in this case PCR 11. This allows a remote user to seal to the state of PCR 11, and make data unavailable should an administrator log in as root. This functionality was not implemented in the prototype, but would be straightforward. All administration must be through SSH or terminal log in. Login can be detected through IMA extending the 'pam' authentication modules.

```
/*@ requires message != null;
  @ ensures tbb.getCurrentVotes().size() >
            \old(tbb.getCurrentVotes().size()); */
public void addVote(String message) { ... }

/*@ requires nonce != null && nonce.length >= 20;
  @ ensures (\result != null) && (\result.getAttestation() != null);
  @ ensures \result.getTickCount() > \old(getCurrentMessages().size()); */
public BoardLogAttestation attest(byte[] nonce) {...}

/*@ ensures \result != null; */
public /*@ pure @*/ List getCurrentVotes() {...}

/*@ requires start != null && end != null;
  @ ensures historyComplete( \result, start, end ); */
public List getHistory(Date start, Date end) {...}

public /*@ pure @*/ static boolean historyComplete(
    List history, Date start, Date end) {
  long startVal, endVal = -1;
  long counterLabel, msgCount = 0;
  for (int i=0; i< history.size(); i++) {
    BoardLogSaveFile h = (BoardLogSaveFile) history.get(i);
    if (h.getEnd() != null && h.getEnd().before(start))
      return false; //history item ended before the start time
    if (h.getStart() != null && h.getStart().after(end))
      return false; //history item began after the end time
    if (i==0) {
      startVal = h.getCounterStartVal();
      endVal = h.getCounterVal();
      counterLabel = h.getCounterLabel();
    } else {
      if (h.getCounterStartVal() != endVal)
        return false; // counter has skipped an item in the history
      if (h.getCounterLabel() != counterLabel)
        return false; // counter has changed label
      endVal = h.getCounterVal();
    }
    msgCount += h.getAttestation().getBoardEntries().size();
    if (msgCount != (endVal - startVal))
      return false; //total count of votes is wrong
  }
  return true;
}
```

Figure 8.2: Code extracts demonstrating the JML in the Trusted Ballot Box service

8.2.5 Integrity reports

The running back-end platform, containing both services, has a total of 206 measurements. As a point of comparison, this is 71 fewer than the service implemented in Section 4.2.2. This platform was running two RMI services, the TPDMenu shell, accessed through SSH. Before attesting, it had been queried by a remote platform to add a ballot to its store.

8.2.6 Verification model

The verification process includes use of the modelling approach defined in Chapter 6. This requires a model for each program on the platform, all of which are combined to create a full platform model. The following model is for the ballot box service itself. It shows how the service transitions, and how PCRs are modified. It interacts with the JRE model found in Section 6.4.1.

The launch script is defined as `ws-script` and launches the JRE application, with the configuration defined in states `ws-script` and `ws-java`. The script itself will also be measured as it is defined as a programme. The first state defined in `ws-script` overrides the JRE's default classpath to add the service archive. Recall that a non-zero third argument in `state` definitions shows an overriden state:

```
programme(
   ws-script, [app(jre,[ws-script,ws-java])]
).
state(  ws-script, jre-loadclasses, 50, 'service.jar',
        end
).
```

The behaviour of the service is defined in the following states. The last state is idle, and will transition depending on the input received:

```
state(  ws-java, jre-app-start, 50, 'Ballot box start',
        newstate(ws-bb-checkpoint)
).
state(  ws-java, ws-bb-checkpoint, 0, 'Checkpoint',
        newstate(ws-bb-checkpoint2)
).
state(  ws-java, ws-bb-checkpoint2, 0, 'TimeStamp',
        newstate(ws-bb-init)
).
state(  ws-java, ws-bb-init, 0, 'Ballot box waiting for input',
        choice([ newstate(ws-bb-receive),
                 newstate(ws-bb-shutdown),
                 newstate(ws-bb-checkpoint)])
).
```

Finally, the next two states define what happens when a ballot is received. Ballots are measured as `ballot(...)`, allowing any input in this format. However, this can be translated in the real validation to match whatever form the ballots actually take.

```
state(  ws-java, ws-bb-receive, 0, ballot(X),
        newstate(ws-bb-init)
).
state(  ws-java, ws-bb-shutdown, 0, 'Checkpoint',
        newstate(jre-end)
).
```

The example used in Section 6.4.1 could be modified for the rest of the programs on the platform. The main assumptions made when using these models is that the code is accurately described by the model, and that no runtime attacks occur.

Having defined the behavioural model, the integrity measurement log can be interpreted by it to check that no unexpected state transition occurs – e.g. every checkpoint has a new timestamp – and the resulting view of platform state can be used to count the number of ballots received and even extract their content.

8.2.7 Verifying the vote tally

The overall verification procedure requires a complete list of records for the whole period of the election. Because the election system may have shutdown and restarted on occasion, there may be multiple attestations and logs to verify. The following artefacts are needed in this process:

- The AIK certificate of the platform: $AIKCredential_{SK(PCA)}\{|\ AIK\text{-}PK(B)_1\ |\}$.
- A list of voting *records* for each time the election system has been in used. Each record consists of:
  - An integrity measurement log, showing which hashes were extended to pcr_{0-11}. Examples of the IML can be seen in Section A.3.8.
  - A TPM Quote: $Quote_{AIK-SK(B)_1}\{|\ pcr_{0-11}, nonce\ |\}$.
  - A signed monotonic counter value.

The first step in the process is to check that all AIK credentials are signed by a trusted Privacy CA, each credential is still valid, and that the same AIK has been used to sign counters and TPM Quotes. Then the TPM Quote can be compared to the IML, to check that the hash chain matches the PCR value. This verifies that the logs are a genuine account of PCR actions. Freshness is guaranteed by the final record, which will contain a nonce set by the challenger. Subsequent records are not fresh, but the counter value can be used to order them.

Starting at the first record (with the lowest counter value), the following algorithm checks that no votes are missing. For each record $record_n$:

1. Run IML_n through the model-runner, using a set of component models already obtained. This shows that the behaviour of the platform followed the behaviour of the specified programs.
2. Make sure that all of the programs that have been run are considered trustworthy.
3. Identify the first ballot ($ballot_n^0$) from IML_n, and the counter value associated with it ($count_n^0$).
4. Check that for each subsequent counter value ($count_n^1, count_n^2, ...$), another entry has been extended into the PCRs and in the log.
5. At the end of the log, check that the number of votes (v) recorded equals the final counter value ($count_n^v$), less the original counter value ($count_n^0$).
6. Find the next log (IML_{n+1}) and record, and check that it begins with a counter value equal to the end counter value of the previous log: $count_{n+1}^0 = count_n^v$.

8.2.8 Verifying the software

The final step is to check that the voting software is trustworthy. If not, it could invent spurious votes or change the value of the vote entered. This is achieved through trustable remote verification as outlined in Chapter 7.2. The key assumptions are that the compilation tools are trustworthy (Ant, JMLC, Java) and that the JML annotations demonstrate suitable properties.

8.2.9 Unimplemented features

Some necessary features remain unimplemented. This is because of time constraints and the fact that some of the less novel programs were unlikely to contribute to the evaluation process. Most importantly, messages to each service should be encrypted with a TPM key, as defined in Section 5.2.2. This involves modifying the web service middleware, a time-consuming (although probably not too complicated) process. Signing of the results was similarly left out. Another required modification is to make the back-end platform measure additional runtime events. The login application needed to measure files such as `/etc/passwd` and `/etc/shells` , and invalidate PCRs on root login. From the experience of developing TPDMenu, this should not involve many lines of code.

For the compilation phase, Ant scripts support PCR usage, but the Ant interpreter itself needed to be modified to properly support integrity measurement. However, much of the work can be done by the trusted JVM that Ant runs on.

TPDMenu was modified as described in Section 6.5. However, running on the target platform was a problem, as it used an incompatible TPM library. For the purpose of the evaluation, its behaviour was recorded on a different machine. Finally, attestations of monotonic counter values were not implemented, due to limitations with the TPM libraries.

8.2.10 Summary

The ballot box implementation demonstrates that the methods proposed in the rest of this thesis are practical and possible to use. It also shows how much complexity is required during the assurance process. While it is impossible to state *exactly* how strong the assurances are, and how trustworthy the service is, it is clear that it takes relatively little additional work to use these ideas. The rest of this chapter will discuss how much assurance can be gained from attesting this platform, and therefore how feasible the use of attestation is in general.

8.3 To What Extent Have Attestation Problems Been Solved?

Several of the problems discussed in Chapter 3.1 have been solved. The main improvement is with respect to the *semantic gap* problem. Because code properties can be specified and attested to, it is now possible to gain assurance in platform behaviour, beyond simple whitelisting of executables. The ballot box service can show that it will increment its counters on every received ballot, and that it will report a log of events from the correct time period. Furthermore, by taking the system modelling approach described in Chapter 6 the overall system state can be analysed and the measurement and reporting of any background events (such as administration) will not come as a surprise to the challenger. In some scenarios this even means that the range of future events can be limited. It is also true that the attested platform has relatively more code running that is interesting to the user, as the middleware resides on the untrusted front-end. This makes attestation more meaningful with respect to the security concerns of the challenger.

The chance of runtime attack has also been reduced. Because all of the possible integrity measurements are fully listed in the model, any *real* attack that causes an unexpected measurement can be taken seriously without potential for a false positive. The split architecture also hardens the platform, greatly reducing the amount of trusted code running. The only open ports are for the RMI communication and SSH, and no parsing of XML is necessary. The additional runtime checks introduced by JMLC can be considered a defence against code-level attacks. Pre-conditions potentially reduce the chance of null-pointer or injection-style attacks, as these are specified and automatically rejected.

Support for whitelisting has also improved. Attesting a web service is more realistic than many individual client machines (full discussion in Section 4.1.1) and with a split architecture results in at least 30% fewer whitelist entries. The concept of a whitelist is also enhanced by the program models. These help to solve related configuration issues, as program models serve as a better abstraction than simple file hashes. Whitelists can also be made more complete, including any of the libraries specified in the compilation process. This additional *provenance* is an unintended advantage of the approach. In the ballot box example, there were only 206 measurements on the back-end platform, all of which were either standard components (kernel modules, command line applications), relatively simple scripts that could have their behaviour modelled, or the custom-made and compiled web services.

However, some problems remain. The same number of *trusted parties* are required as in any

attestable system, although it is worth noting that no extra parties are needed, despite providing more useful levels of assurance. Privacy has not been improved, although the service owner is now free to use any software on the front-end platform, as this does not require measurement. Unfortunately, the compilation certificate also decreases privacy by declaring the libraries and code used to compile the application, as well as those used to run it. These issues are less important for the electronic voting example. Some runtime issues remain, mainly because the operating system and libraries are likely to have vulnerabilities. Complementary platform hardening techniques would go a long way to improving this. Another issue is that not all behaviour can be captured by this approach due to JML limitations. However, this may be mitigated through additional verification tools. The *trusted path* and *multiple domain* issues have not been considered.

Success with respect to compatibility with legacy systems is mixed. The JVM and build process required modification, and all running programs must now provide PCR usage models as well as hashes. Any programs in the TCB must also be modified to support measure-before-load. This is a significant practical overhead. However, moving a lot of the software to the front-end platform helps reduce the effort. Furthermore, the runtime models can be produced independently of the programs themselves.

Performance is an area that has suffered. The split-service approach is slow, as outlined in Section 5.5, but may be considered a reasonable trade-off. Interpreting attestations with the modelling approach is not necessarily performance-intensive, although this depends largely on the model. As discussed in Sections 6.3.5 and 6.4.4, complex models featuring cached results or recursive behaviour can be slow, which will have an impact every time an attestation is interpreted. Remote verification, on the other hand, is a one-off event which has no significant performance penalty in theory. In practice adding layers of indirection (as the JML compiler does) can reduce service performance. A static ESC/Java2 analysis would have no adverse runtime impact.

8.4 Practicality and Security of Solutions

While each new solution has been analysed in the earlier chapters, this section will discuss the practical overheads.

Firstly, the additional assurance comes at the cost of trusting the libraries and tools in use. Trustable remote verification requires the user to trust a significant number of tools, including an operating system, Java, Ant, the JML compiler, and other libraries. It will be difficult to assess all of these. However, because the overall approach is tool-agnostic, this can be reduced in return for a less sophisticated compilation setup. The same is true for the TPM itself – it has been shown to be vulnerable to hardware attacks [205] – but any brand of TPM can be used, and multiple can be used for the compilation certificate.

The next overhead is the complexity on the user. The certificates require effort to validate, including both the Ant scripts and JML annotations. This essentially requires users to have prior training in Java. Another task for verifiers is to interpret the result of running a platform

model. Running a model will only produce a view of system state, and potential future behaviour. Input is still required to decide on trustworthiness. However, both of these requirements on the user could be reduced given further automation. A fair analysis is that the system described in this chapter could be assessed by a well-programmed application on behalf of the user, but is far from understandable for most users. This additional program is another component that must be trusted by the challenger.

8.5 Assurance Properties

The assurance goals described in Section 8.2.1 rely on the more general assurance properties discussed in Section 2.5. The important properties are that *known* software can be whitelisted, *unknown* service software can have its behaviour securely established, runtime events are recorded (in this case, votes being received) and runtime attacks do not occur. Having established that attestation problems have been reduced, what is the impact on these properties in particular?

Whitelisting requires the relying party to have an up-to-date list of trustworthy software. In the case of this ballot box, this is achievable, as most software comes from the Ubuntu Linux repository. The service itself does not, but the compilation credential can provide the provenance for this application. As only around 200 entries are required on the measurement log, this is not a problem.

Runtime events must be recorded properly. This is asserted in the source code of the service, which uses PCR values and TPM Quotes to attest to votes being received. This can be verified through the JMLC annotations, but more strongly through use of the TPM counters. The runtime model also makes sure that no other software could be intercepting votes. It also makes it easy to check that the PCR values match a sensible platform state (e.g., one that was capable of accepting votes when it did, and that extended a PCR for every vote received). Hardware errors, or attacks on availability mean that each vote submitted may not be recorded, but should a vote be accepted and recorded, the PCR model and TPM counters will make later denial impossible. JML annotations can be used to make sure the vote itself has not changed.

Runtime attacks should not occur. While it is not possible to prove this, some measures do make this a more reasonable assumption (or, at least a more expensive attack to perform). The attack surface of the ballot box is small – only the RMI service and an SSH daemon are listening for input on the network – and it might be considered reasonable to trust both of these. Assuming the RMI service and JVM are trustworthy, there could be a further problem in the service itself. However, the service runs in managed code, and the JML annotations will produce an error if pre-conditions on the input are not met (such as a null pointer). They could be further elaborated to specify input lengths. This is a more trustworthy scenario than relying on an XML and SOAP interpreter, and an improvement in this assurance property.

8.6 Is Attestation Feasible for Service Assurance?

Attestation is known to have many problems as an assurance mechanism, and greater analysis does not make these issues disappear. Rather, it has helped focus what the issues are, and how they might be avoided.

The first key point is that the whitelisting problem is much smaller in service-oriented architectures, as demonstrated in Section 4.2. With only around 280 components, going down to 206 with the removal of middleware, this seems practical. However, validating 200 components is still difficult. The complexity of modern operating systems, the behaviour of which may not be trusted in the face of potential runtime attacks, is the root problem. Eliminating some complex software has helped, but must go further.

Another criticism is that PCR values are so far from platform behaviour, that the wrong information is being provided for assurance. However, the combination of a better defined integrity measurement process, as well as verification and compilation certificates, means that users can have great confidence in the behaviour of remote software. This also comes without many of the overheads associated with runtime reporting systems. However, there are still ways in which these systems could be compromised. A runtime attack to the verification stage, or to one of the running programs would invalidate many of the assumptions. Furthermore, the compilation certificates rely heavily on tools which were not necessarily designed with integrity in mind. If the compilers cannot be trusted to protect against dishonest use, the behavioural assumptions fail. The distance between attestation and assurance has been closed, but at the cost of reliance on more trusted components.

The electronic voting system implemented for this evaluation is an example of both the progress made and the problems remaining. The voting platform is single-purpose, and this purpose is strongly linked to the implementation through JML annotations. However, the best way of subverting the system is still through exploiting either the operating system, Java runtime libraries, or by writing difficult-to interpret JML annotations which do not make the guarantees they appear to. It is the *tools* and *runtime environment* that remain as weak points. Attestation will not be able to make the potentially strong guarantees that it is capable of until these monolithic programs become more trustworthy.

Chapter 9

Conclusion and Future Work

This dissertation has investigated the extent to which TCG-defined attestation is a suitable mechanism for assurance of remote services. During the analysis, several methods have been developed to improve service attestation. These are discussed in Section 9.1. In addition to these, Section 9.2 outlines several future opportunities and open problems to explore. Finally, Section 9.3 concludes by summarising the contributions of the dissertation and answering the thesis question.

9.1 Contributions

9.1.1 Attestation analysis and measurement

The first significant contribution of this dissertation is the thorough analysis of integrity measurement and attestation. Section 3.1 presents a break-down of the current problems, Section 3.3.1 shows a taxonomy of measurable components, and Section 3.3 covers all existing (and some new) methods of measuring them. In addition, the analysis in Section 4.1 presents a set of principles to guide where attestation should work best, and how this relates to web services. The conclusion is that servers, rather than clients, should be considered a more feasible target for integrity reporting. This is a novel contribution, as most existing literature concentrates either on technical challenges [36, 142], or attempts to apply attestation to a particular scenario [239, 83], without considering its suitability.

Statistics on the difficulty of the well-documented *whitelisting* problem were gathered by attesting a web service platform over a two-and-a-half year period. This is a longer and more in-depth study than any previous work (which usually only included one-time measurement counts) and was analysed to identify how big the problem is, its ongoing scale, and where improvements could be made. This is a contribution in itself, as well as first-hand evidence for the validity of the thesis question.

9.1.2 Reducing the TCB of a web service

The next contribution, presented in Chapter 5, is a technique for minimizing the number of measurements on the platform through isolation of middleware. By splitting a service into two components, it has been shown to reduce the number of integrity measurements without losing functionality. This work highlights a principle that is easy to forget when using attestation: platforms should attest only the software in which the user has, and *should have*, security interest. This is why most of the middleware stack could be removed: the user is only interested in the components which allow access to the service. The other functionality, such as the management interface, load balancing, and auditing, may be unimportant. By removing the middleware, 30% of integrity measurements can be removed from the user's whitelist, and the service itself becomes less vulnerable to attack.

An important part of this contribution was to maintain confidentiality and integrity, without sacrificing standards. Although some compromises were made – a custom verification procedure was necessary – the system works almost entirely within WS-Security standards. This means that the challenger can attest only the core service application, but still communicate with it through a SOAP interface.

9.1.3 Modelling integrity measurement

Chapter 6 presents the next novel contribution: a new approach to verifying integrity measurement logs which allows for *event reporting* in order to gain more information about platform state and behaviour. Existing attestation methods have many problems, either providing guarantees of only execution integrity or relying on a runtime system agent. The proposed alternative is to specify that all programs have simple PCR-usage models. These act as a level of indirection, so that measurement logs can be converted into a more useful description of platform state. State can then be analysed to identify trustworthiness. The main advantages are flexibility and modularity: the system can work with any integrity measurement system, and program models can be composed together independently, allowing for model re-use, and enabling ad-hoc attestation of previously unknown platforms.

In addition to the general approach, two implementations were explored. A CSP modelling system worked well for lower-level programs, and a Prolog tool provided additional flexibility for more complicated usage models. Both of these were developed with a set of design principles in mind, which should help further exploration of this approach. Although there were problems with both implementations, these will serve as a starting point for further model-based approaches. Part of this contribution was also the development of several program models, based on real pieces of software. These serve as useful examples for future work.

9.1.4 Trustable remote verification

Chapter 7 looks at the problem of attesting custom-built software. This is a challenge because of the lack of well-known integrity measurements, unlike off-the-shelf software such as operating

systems. This resulted in a unique approach to *remote verification* – using TPM Quotes (attestations) to certify the build process of a piece of software, along with the results of performing a static analysis on it. This has several theoretical advantages, such as not requiring the relying party to have either the source code or executable, and allowing flexibility of verification and build tools. This compares favourably with other techniques such as proof-carrying code, which rely on the end user having access to the software and running a verifier. As part of this contribution a prototype was built, and a security analysis was performed in order to assess how trustworthy this system could be. Several alternative tools and uses for the approach were also considered.

9.1.5 Applications

As well as the core ideas discussed in this dissertation, several other smaller contributions have been made. Appendix A gives a specification for a trustworthy ballot-box service, which uses the TPM (and principles of append-only attestable memory [43]). This solves a problem highlighted by an analysis of current electronic voting implementation problems [102]. It also was used as part of the evaluation, providing an interesting scenario (with critical security requirements) for testing the techniques developed in this dissertation.

The TPDMenu system (see Section 6.5) was developed as a case study for the investigation of application-level integrity measurement. Based on the existing PDMenu application developed by Joey Hess [89], this menu-based shell is both easy to attest (the menu configuration serving as a whitelist) and restrictive, so that the range of possible administrative tasks can be limited suitably. This approach may be useful for embedded or light weight systems.

9.2 Future Work

There is no shortage of future directions for the topic of this dissertation. As a result, below are just some of the more interesting ideas that could be explored given time. Several other suggestions have already been discussed in Sections 7.5.2 and 6.6.

9.2.1 Policies for integrity measurement models

The modelling approach proposed in Chapter 6 would be greatly enhanced by support for policies. The two implementations allowed for only a visual inspection of platform state, but policies would allow for automated decisions on trustworthiness. For CSP models, this would probably involve defining constraints of the 'afters' of the trace. For example, making sure that it is not possible to execute unknown applications, or that this can only happen if PCRs are invalidated first. This second example might be expressed in a *temporal* logic, such as *linear temporal logic*, which has been shown to be powerful enough to express these kinds of properties [170].

Assuming a policy engine could be built, it would have potential applications to data sealing. Sealing is currently brittle, as any platform update will invalidate any keys sealed to

that level of granularity. Cesena et al. [32] have described a sealing proxy which introduces a level of indirection, to allow for more complex or liberal constraints. This could be integrated with the policy engine to allow for data to be sealed to integrity-measurement polices, rather than just platform state. For example, it could be configured to allow access to data despite a background task extending a PCR value, but still prevent data access depending on fine-grained application behaviour.

9.2.2 Attestable and minimal operating systems

Chapter 5 concentrates on removing middleware from the trusted computing base of a service, but more significant impact would be had by removing the operating system. Figure 4.1 shows that the kernel is the biggest source of updates and measurements. As has already been discussed, removing middleware does help: it allows for a smaller operating system, requiring fewer features and drivers. However, this dissertation stopped short of investigating how to minimize the OS with this in mind.

A smaller, more trustworthy operating system could improve the security of systems such as trustable remote verification, as well as reducing the cost of attesting an individual service instance. With the popularity of cloud computing, which allows the dynamic creation of new virtual machines, assurance of the operating system seems particularly important. Some of the opportunities for minimization include reducing the number of communication channels, and unnecessary separation of OS and applications. If code only exists on the platform to communicate via one piece of hardware, and one protocol, this would reduce code rather than introduce complex firewall rules, as is the norm today. Furthermore, if isolation is provided by the hypervisor or virtual machine monitor, and the virtual machine only provided one application, this would allow for the elimination of OS versus user-space isolation. As vulnerabilities allowing privilege escalation are frequently discovered in modern operating systems, this would remove complexity without loss of security. For future work, looking into the customised delivery of extremely low-complexity operating systems could greatly increase the usefulness of attestation.

9.2.3 Inter-service attestation and communication

One of the issues that has been identified repeatedly in this dissertation is that online services tend to be used in collaboration with many others, and some are *composite services* which provide a simple front-end to a whole network of component services. As the evaluation has demonstrated, this is a problem for some of the solutions discussed in Chapter 5. Furthermore, services that rely on the results of others cannot be checked for code-correctness, making trustable verification less useful. A solution to the problem of verifying multiple services would be the natural next step in this project.

There are a number of potential solutions. As proposed in Section 7.4.4, the verification problem might be solved by presenting the user with a whole set of certificates for each service. However, this would be a problem for dynamic selection and usability. How would

the complete set of services be known in advance? Would users be expected to verify every possible service? The alternative might be for users to verify the front-end's ability to select services which will fulfil the required constraints. This would be a challenging area for future work.

9.3 Summary

This dissertation documents an investigation into the practicality of TCG-defined attestation, and the extent to which it is a feasible mechanism for obtaining assurance of the trustworthiness of remote services. This topic is inspired by the increasing need for trustworthy services, and the potential suitability of attestation. In principle, integrity reporting, in combination with other software assurance techniques, ought to provide exactly the assurances necessary when using services with sensitive, security-critical data. However, several problems with attestation have already been identified, including the effort required to maintain software whitelists, the difference between reported execution state and security state, and the lack of protection against runtime attacks. This dissertation questions the extent to which these issues are a problem for assurance of service providers – rather than clients – and proposes solutions to some of them.

Three distinct solutions have been presented. The first aimed to solve the software whitelisting problem by reducing the size of integrity measurement logs, through removal of service middleware. This resulted in a 30% decrease in measurements over time, as demonstrated by the comprehensive analysis of a service platform over a two and a half year period. Perhaps more significantly, the proposed two-tier architecture also increased the security of the platform, without losing middleware functionality or conformance with web service standards.

The second solution was to improve the relationship between attestation and platform state, through modelling of programs. The principle is that programs can report more information about their current state through extending extra information into PCRs. To manage the verification of these more complex interactions with the TPM, programs must declare how they intend to use PCRs, and how integrity reports reflect their behaviour. This overcomes limitations in the current state of the art – systems like IMA, and the TCG Platform Trust Service – and provides a general-purpose model for any attesting application, as shown by several examples. Although there is still room for improvement, this approach provides a solid starting-point for future investigations.

Finally, the *trustable remote verification* approach was invented in order to solve the problem of attesting custom, unknown executables. This is a particular issue in service-oriented architectures, as each service endpoint may offer unique functionality. By linking the attested binaries to a contract enforced at compile-time, even unknown executables can be trusted to behave within certain limits. This approach is tool-agnostic, and has potentially no overhead on the service endpoint, unlike many alternatives.

These techniques were combined and used to develop an online ballot box, a key component of an electronic voting system. The successful implementation of this provided validation of

these ideas, and served as a suitable target for evaluation. This identified that attestation *has become a more practical technique*, and is more appropriate for service assurance with these new contributions. However, the limitations are now largely due to the untrustworthy nature of many of the tools used, including the operating system and standard libraries. The potential for runtime compromise of these large components means that attestation remains less practical than it should be.

In conclusion, attestation *can* provide service assurance, and should be considered an appropriate way of assessing the trustworthiness of remote servers. It is entirely plausible to use attestation for whitelisting web service software, particularly using the three techniques outlined here. This will provide users with a guarantee of behaviour, and make it significantly more difficult for malicious insiders, or outside attackers, to subvert service-oriented systems. However, the levels of assurance provided may not reach the *highest* level due to the untrustworthy nature of operating systems and runtime applications. With these components improved, attestation could provide the foundation for a wide range of sophisticated program analysis-based assurance techniques.

Bibliography

[1] Martín Abadi and Ted Wobber. A Logical Account of NGSCB. In David de Frutos-Escrig and Manuel Núñez, editors, *FORTE '04: Proceedings of Formal Techniques for Networked and Distributed Systems*, **3235** of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2004.

[2] M. Alam, J.-P. Seifert, and Xinwen Zhang. A Model-Driven Framework for Trusted Computing Based Systems. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, pages 75–75, October 2007.

[3] Masoom Alam, Xinwen Zhang, Mohammad Nauman, and Tamleek Ali. Behavioral attestation for web services (BA4WS). In *SWS '08: Proceedings of the 2008 ACM workshop on Secure Web Services*, pages 21–28, New York, NY, USA, 2008. ACM.

[4] Masoom Alam, Xinwen Zhang, Mohammad Nauman, Tamleek Ali, and Jean-Pierre Seifert. Model-based behavioral attestation. In *SACMAT '08: Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, pages 175–184, New York, NY, USA, 2008. ACM.

[5] Sami Alsouri, Özgür Dagdelen, and Stefan Katzenbeisser. Group-based Attestation: Enhancing Privacy and Management in Remote Attestation. In Sean W. Smith Alessandro Acquisti and Ahmad-Reza Sadeghi, editors, *TRUST 2010: Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, **6101/2010** of *Lecture Notes in Computer Science*, pages 63–78. Springer, June 2010.

[6] E. Amoroso, T. Nguyen, J. Weiss, J. Watson, P. Lapiska, and T. Starr. Toward an approach to measuring software trust. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy.*, pages 198–218, Oakland, CA , USA, May 1991. IEEE.

[7] Melvin J. Anderson, Micha Moffie, and Chris I. Dalton. Towards Trustworthy Virtualisation Environments: Xen Library OS Security Service Infrastructure. Technical Report HPL-2007-69, HP Laboratories Bristol, April 2007.

[8] Andrew W. Appel, Neophytos Michael, Aaron Stump, and Roberto Virga. A Trustworthy Proof Checker. *Journal of Automated Reasoning*, **31**(3-4):231–260, 2003.

[9] Robert Atkey, Kenneth MacKenzie, and Christopher Paton. Secure Execution of Mobile Java using Static Analysis and Proof Carrying Code. In Simon J Cox, editor, *Proceedings of the UK e-Science All Hands Conference 2007*, Nottingham, UK, September 2007. National e-Science Centre.

[10] Fabrizio Baiardi, Diego Cilea, Daniele Sgandurra, and Francesco Ceccarelli. Measuring Semantic Integrity for Remote Attestation. In Liqun Chen, Chris J. Mitchell, and Andrew Martin, editors, *TRUST '09: Proceedings of the Second International Conference on Trusted Computing*, **5471** of *Lecture Notes in Computer Science*, pages 81–100, Oxford, UK, April 2009. Springer.

[11] Shane Balfe and Anish Mohammed. Final Fantasy - Securing On-Line Gaming with Trusted Computing. In Bin Xiao, Laurence Tianruo Yang, Jianhua Ma, Christian Müller-Schloer, and Yu Hua, editors, *ATC 07: Proceedings of the 4th International Conference on Autonomic and Trusted Computing*, **4610** of *Lecture Notes in Computer Science*, pages 123–134. Springer, July 2007.

[12] Endre Bangerter, Maksim Djackov, and Ahmad-Reza Sadeghi. A Demonstrative Ad Hoc Attestation System. In Vincent Rijmen Tzong-Chen Wu, Chin-Laung Lei and Der-Tsai Lee, editors, *ISC '08: Proceedings of the 11th International Conference on Information Security*, **5222** of *Lecture Notes in Computer Science*, pages 17–30, Taipei, Taiwan, September 2008. Springer.

[13] V.R. Basili, L.C. Briand, and W.L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, **22**(10):751–761, October 1996.

[14] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: virtualizing the trusted platform module. In *USENIX-SS'06: Proceedings of the 15th USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2006. USENIX Association.

[15] Agreiter Berthold, Muhammad Alam, Ruth Breu, Michael Hafner, Alexander Pretschner, Jean-Pierre Seifert, and Xinwen Zhang. A technical architecture for enforcing usage control requirements in service-oriented architectures. In *SWS '07: Proceedings of the 2007 ACM workshop on Secure Web Services*, SWS '07, pages 18–25, New York, NY, USA, 2007. ACM.

[16] A. Betin-Can and T. Bultan. Verifiable Web Services with Hierarchical Interfaces. In *ICWS'05: Proceedings of the IEEE International Conference on Web Services*, **1**, pages 85–94. IEEE, July 2005.

[17] Aysu Betin-Can, Tevfik Bultan, and Xiang Fu. Design for verification for asynchronously communicating Web services. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 750–759, New York, NY, USA, 2005. ACM.

[18] Nishchal Bhalla and Sahba Kazerooni. Web Service Vulnerabilities. Presented at Black Hat Europe 2007 http://www.blackhat.com/presentations/bh-europe-07/Bhalla-Kazerooni/Whitepaper/bh-eu-07-bhalla-WP.pdf, February 2007.

[19] Rajendra Bose and James Frew. Lineage retrieval for scientific data processing: a survey. *ACM Computing Surveys*, **37**(1):1–28, March 2005.

[20] Alexander Böttcher, Bernhard Kauer, and Hermann Härtig. Trusted Computing Serving an Anonymity Service. In Peter Lipp and Koch [163], pages 143–154.

[21] Andrea Bottoni, Gianluca Dini, and Evangelos Kranakis. Credentials and Beliefs in Remote Trusted Platforms Attestation. In *WOWMOM '06: Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*, pages 662–667, Washington, DC, USA, 2006. IEEE Computer Society.

[22] Brent Boyer. Java benchmarking. http://www.ellipticgroup.com/html/benchmarkingArticle.html, June 2008.

[23] Uri Braun, Avraham Shinnar, and Margo Seltzer. Securing provenance. In *HOTSEC'08: Proceedings of the 3rd conference on Hot topics in security*, pages 1–5, Berkeley, CA, USA, 2008. USENIX.

[24] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *CCS '04: Proceedings of the 11th ACM conference on Computer and Communications Security*, pages 132–145, New York, NY, USA, 2004. ACM.

[25] Gerald Brose. Securing Web Services with SOAP Security Proxies. http://www.xtradyne.com/documents/whitepapers/Xtradyne-WebServices_Security_Proxies.pdf, 2004.

[26] J. R. Brown and R. H. Hoffman. Evaluating the effectiveness of software verification: pratical experience with an automated tool. In *AFIPS '72 (Fall, part I): Proceedings of the Fall Joint Computer Conference*, pages 181–190, New York, NY, USA, 1972. ACM.

[27] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Characterizing Kripke Structures in Temporal Logic. In Hartmut Ehrig, Robert A. Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT'87: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, **249** of *Lecture Notes in Computer Science*, pages 256–270. Springer, 1987.

[28] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating Systems Principles*, **23**, pages 1–13, New York, NY, USA, 1989. ACM.

[29] Serdar Cabuk, Liqun Chen, David Plaquin, and Mark Ryan. Trusted integrity measurement and reporting for virtualized platforms. In Liqun Chen and Moti Yung, editors, *INTRUST'09: Proceedings of International Conference on Trusted Systems*, **6163** of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2010.

[30] Damian Carrington. Q&a: 'climategate'. http://www.guardian.co.uk/environment/2010/jul/07/climate-emails-question-answer, July 2010. Newspaper article.

[31] Alessandra Cavarra and James Welch. Behavioural Specifications from Class Models. In Jim Davies and Jeremy Gibbons, editors, *IFM 07: Proceedings of the 6th International Conference on Integrated Formal Methods*, **4591** of *Lecture Notes In Computer Science*, pages 118–137, Oxford, UK, July 2007. Springer.

[32] Emanuele Cesena, Gianluca Ramunno, and Davide Vernizzi. Secure storage using a sealing proxy. In *EUROSEC '08: Proceedings of the 1st European workshop on system security*, pages 27–34, New York, NY, USA, 2008. ACM.

[33] Chang Chaowen, He Rongyu, Xie Hui, and Xu Guoyu. A High Efficiency Protocol for Reporting Integrity Measurements. In *ISDA '08: Eighth International Conference on Intelligent Systems Design and Applications*, **2**, pages 358–362. IEEE, Nov. 2008.

[34] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *CCS '02 Proceedings of the 9th ACM conference on Computer and Communications Security*, pages 235–244. ACM, 2002.

[35] Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble. A Protocol for Property-based Attestation. In *STC '06: Proceedings of the First ACM Workshop on Scalable Trusted Computing*, pages 7–16, New York, NY, USA, 2006. ACM.

[36] Liqun Chen, Hans Löhr, Mark Manulis, and Ahmad-Reza Sadeghi. Property-Based Attestation without a Trusted Third Party. In Vincent Rijmen Tzong-Chen Wu, Chin-Laung Lei and Der-Tsai Lee, editors, *ISC '08: Proceedings of the 11th international conference on Information Security*, **5222** of *Lecture Notes in Computer Science*, pages 31–46, Berlin, Heidelberg, 2008. Springer.

[37] Liqun Chen, Chris J. Mitchell, and Andrew Martin, editors. *Trust '09: Proceedings of the Second International Conference on Trusted Computing*, **5471** of *Lecture Notes in Computer Science*. Springer, April 2009.

[38] James Cheney, editor. *TaPP '09: Proceedings of the First Workshop on the Theory and Practice of Provenance*. USENIX, 2009.

[39] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, **1**(4):379–474, 2009.

[40] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, **2**(6):76–79, November 2004.

[41] B.V. Chess. Improving computer security using extended static checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 160–173. IEEE, 2002.

[42] SuGil Choi, Jin-Hee Han, and Sung-Ik Jun. Improvement on TCG Attestation and Its Implication for DRM. In Osvaldo Gervasi and Marina L. Gavrilova, editors, *ICCSA '07: Proceedings of the International Conference on Computational Science and Its Applications*, **4705** of *Lecture Notes in Computer Science*, pages 912–925. Springer, 2007.

[43] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating Systems Principles*, **41**, pages 189–204, New York, NY, USA, 2007. ACM.

[44] M.R. Clarkson, S. Chong, and A.C. Myers. Civitas: Toward a Secure Voting System. In *S&P '08: Proceedings of the IEEE Symposium on Security and Privacy*, pages 354–368. IEEE, May 2008.

[45] Ben Clifford, Ian Foster, Jens-S. Voeckler, Michael Wilde, and Yong Zhao. Tracking provenance in a virtual data grid. *Concurrency and Computation: Practice and Experience*, **20**(5):565–575, 2008.

[46] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS '04: Proceedings of the International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, **3362** of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

[47] George Coker, Joshua D. Guttman, Peter Loscocco, Justin Sheehy, and Brian T. Sniffen. Attestation: Evidence and Trust. In Liqun Chen, Mark Dermot Ryan, and Guilin Wang, editors, *ICICS '08: Proceedings of the 10th International Conference on Information and Communications Security*, **5308** of *Lecture Notes in Computer Science*, pages 1–18, Birmingham, UK, 2008. Springer.

[48] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for Java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation*, pages 95–107, New York, NY, USA, 2000. ACM.

[49] Lionel Cons, Karl Berry, and Olaf Bachmann. *ProBE User Manual*. Formal Systems (Europe) Ltd, March 2003.

[50] A. Cooper and A. Martin. Towards a secure, tamper-proof grid platform. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, **1**, page 8. IEEE, May 2006.

[51] Andrew Cooper. *Towards a Trusted Grid Architecture*. PhD thesis, Oxford University Computing Laboratory, 2009.

[52] M. A. Crook. The Caldicott report and patient confidentiality. *Journal of Clinical Pathology*, **56**(6):426–428, 2003.

[53] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A Logic of Secure Systems and its Application to Trusted Computing. In *S&P '09: Proceedings of the 30th IEEE Symposium on Security and Privacy*, pages 221–236. IEEE, May 2009.

[54] Jim Davies, James Welch, Alessandra Cavarra, and Edward Crichton. On the Generation of Object Databases using Booster. In *ICECCS '06: Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems*, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[55] Y. Demchenko, L. Gommans, C. de Laat, and B. Oudenaarde. Web Services and Grid Security Vulnerabilities and Threats Analysis and Model. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 262–267, Washington, DC, USA, 2005. IEEE Computer Society.

[56] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, **52**(3):365–473, May 2005.

[57] Laura DiDio. Server OS Reliability Survey. http://www.iaps.com/2008-server-reliability-survey.html, February 2008.

[58] Kurt Dietrich, Martin Pirker, Tobias Vejda, Ronald Toegl, Thomas Winkler, and Peter Lipp. A Practical Approach for Establishing Trust Relationships between Remote Platforms Using Trusted Computing. In Gilles Barthe and Cédric Fournet, editors, *TGC '07: Proceedings of the Third Symposium on Trustworthy Global Computing*, **4912** of *Lecture Notes in Computer Science*, pages 156–168, Sophia-Antipolis, France, November 2007. Springer.

[59] Edsger W. Dijkstra. Notes on structured programming. In O.-J. Dahl, C. A. R. Hoare, and E. W. Dijkstra, editors, *Structured Programming*, pages 1–82. AP, NY, 1972.

[60] Lo ic Duflot, Daniel Etiemble, and Olivier Grumelard. Using CPU System Management Mode to Circumvent Operating System Security Functions. In *CanSecWest 06: Proceedings of the 2006 CanSecWest Applied Security Conference*, April 2006.

[61] Paul England. Practical Techniques for Operating System Attestation. In Peter Lipp and Koch [163], pages 1–13.

[62] A Ferreira, R Cruz-Correia, L Antunes, P Farinha, E Oliveira-Palhares, D W. Chadwick, and A Costa-Pereira. How to Break Access Control in a Controlled Manner. In *CBMS '06: Proceedings of the 19th IEEE Symposium on Computer-Based Medical Systems*, pages 847–854, Washington, DC, USA, 2006. IEEE Computer Society.

[63] Riccardo Focardi and Fabio Martinelli. A Uniform Approach for the Definition of Security Properties. In Jim Woodcock Jeannette M. Wing and Jim Davies, editors, *FM '99: Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, **1708** of *Lecture Notes in Computer Science*, pages 794–813. Springer, September 1999.

[64] Formal Systems (Europe) Ltd. *FDR2 User Manual*, 1992. http://www.fsel.com/fdr2_manual.html.

[65] M. Franklin, K. Mitcham, S.W. Smith, J. Stabiner, and O. Wild. CA-in-a-Box. In David Chadwick and Gansen Zhao, editors, *EuroPKI '05: Proceedings of the Second European PKI Workshop: Research and Applications*, **3545/2005** of *Lecture Notes In Computer Science*, pages 180–190. Springer, June 2005.

[66] Michael Franz, Deepak Chandra, Andreas Gal, Vivek Haldar, Fermín Reig, and Ning Wang. A portable Virtual Machine target for Proof-Carrying Code. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, Virtual Machines and Emulators*, pages 24–31, New York, NY, USA, 2003. ACM.

[67] James Frew and Peter Slaughter. ES3: A Demonstration of Transparent Provenance for Scientific Computation. In David Koop Juliana Freire and Luc Moreau, editors, *IPAW 08: Proceedings of the Second International Provenance and Annotation Workshop*, **5272** of *Lecture Notes in Computer Science*, pages 200–207, Salt Lake City, UT, USA, June 2008. Springer.

[68] Ryan Gardner, Sujata Garera, and Aviel D. Rubin. On the Difficulty of Validating Voting Machine Software with Software. In *EVT '07: Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop*, Boston, MA, August 2007. USENIX.

[69] Simson Garfinkel, Gene Spafford, and Alan Schwartz. *Practical UNIX and Internet Security*, chapter 3, page 35. O'Reilly, 3rd edition, 2003.

[70] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS '03: Proceedings of the Network and Distributed Systems Security Symposium*, pages 191–206. The Internet Society (ISOC), 2003.

[71] Tal Garfinkel, Mendel Rosenblum, and Dan Boneh. Flexible OS Support and Applications for Trusted Computing. In Michael B. Jones, editor, *Proceedings of HotOS '03: the 9th Workshop on Hot Topics in Operating Systems*, pages 145–150, Lihue (Kauai), Hawaii, USA, 2003. USENIX.

[72] Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, and N. Asokan. Beyond secure channels. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable Trusted Computing*, pages 30–40, New York, NY, USA, 2007. ACM.

[73] Yolanda Gil, Ewa Deelman, Mark Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis Gannon, Carole Goble, Miron Livny, Luc Moreau, and Jim Myers. Examining the Challenges of Scientific Workflows. *IEEE Computer*, **40**(12):26–34, December 2007.

[74] Kenneth Goldman, Ronald Perez, and Reiner Sailer. Linking remote attestation to secure tunnel endpoints. In *STC '06: Proceedings of the first ACM workshop on Scalable Trusted Computing*, pages 21–24, New York, NY, USA, 2006. ACM.

[75] Dieter Gollmann. Why Trust is Bad for Security. In *STM '05: Proceedings of the First International Workshop on Security and Trust Management*, **157** of *Electronic Notes in Theoretical Computer Science*, pages 3 – 9. Elsevier, May 2006.

[76] Dan Goodin. Feds: IT admin plotted to erase Fannie Mae. The Register, http://www.theregister.co.uk/2009/01/29/fannie_mae_sabotage_averted/, January 2009.

[77] T. Grandison and M. Sloman. A survey of trust in Internet applications. *IEEE Communications Surveys and Tutorials*, **3**:2–16, January 2001.

[78] David Grawrock. *Dynamics of a Trusted Platform*, chapter 13, pages 198–199. Intel Press, 2009.

[79] David Grawrock. *Dynamics of a Trusted Platform*. Intel Press, February 2009.

[80] N. A. B. Gray. Comparison of web services, java-rmi, and corba service implementation. In Jean-Guy Schneider and Jun Han, editors, *AWSA '04: Proceedings of the Fifth Australasian Workshop on Software and System Architectures*, pages 52–63. Swinburne University of Technology, 2004.

[81] P. Groth and L. Moreau. Recording Process Documentation for Provenance. *IEEE Transactions on Parallel and Distributed Systems*, **20**(9):1246–1259, September 2009.

[82] Paul Groth, Sheng Jiang, Simon Miles, Steve Munroe, Victor Tan, Sofia Tsasakou, and Luc Moreau. An Architecture for Provenance Systems. Technical Report 13216, University of Southampton School of Electronics and Computer Science, November 2006.

[83] Liang Gu, Xuhua Ding, Robert Huijie Deng, Bing Xie, and Hong Mei. Remote attestation on program execution. In *STC '08: Proceedings of the 3rd ACM workshop on Scalable Trusted Computing*, pages 11–20, New York, NY, USA, 2008. ACM.

[84] Vivek Haldar. *Semantic Remote Attestation*. PhD thesis, University of California, Long Beach, CA, USA, 2006. Adviser-Michael Franz.

[85] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic Remote Attestation - Virtual Machine Directed Approach to Trusted Computing. In *VM '04: Proceedings of the Third Virtual Machine Research and Technology Symposium*, pages 29–41. USENIX, 2004.

[86] Ragib Hasan, Radu Sion, and Marianne Winslett. Introducing secure provenance: problems and challenges. In *StorageSS '07: Proceedings of the 2007 ACM workshop on Storage Security and Survivability*, pages 13–18, New York, NY, USA, 2007. ACM.

[87] Ragib Hasan, Radu Sion, and Marianne Winslett. The case of the fake Picasso: preventing history forgery with secure provenance. In *FAST '09: Proccedings of the 7th conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2009. USENIX.

[88] R. Heckel and M. Lohmann. Towards Contract-based Testing of Web Services. In Mauro Pezzé, editor, *TACoS '04: Proceedings of the International Workshop on Test and Analysis of Component Based Systems*, **116** of *Electronic Notes in Theoretical Computer Science*, pages 145–156. Elsevier, 2005.

[89] Joey Hess. PDMenu Website. http://kitenet.net/~joey/code/pdmenu/, August 2009.

[90] Mayumi Hori and Masakazu Ohashi. Applying XML Web Services into Health Care Management. In *HICSS'05: Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, **6**, page 155a, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[91] Luke Hornof and Trevor Jim. Certifying Compilation and Run-Time Code Generation. *Higher-Order and Symbolic Computation*, **12**(4):60–74, December 1999.

[92] Jun Ho Huh, John Lyle, Cornelius Namiluko, and Andrew Martin. Managing application whitelists in trusted distributed systems . *Future Generation Computer Systems*, **27**(2):211–226, February 2011.

[93] Jun Ho Huh and A. Martin. Trusted Logging for Grid Computing. In *APTC '08: Proceedings of the Third Asia-Pacific Trusted Infrastructure Technologies Conference*, pages 30 –42. IEEE, 14-17 2008.

[94] Marty Humphrey and Mary R. Thompson. Security Implications of Typical Grid Computing Usage Scenarios. *Cluster Computing*, **5**(3):257–264, 2002.

[95] Intel. Statistical analysis of floating point flaw: Intel white paper. http://www.intel.com/support/processors/pentium/fdiv/wp/, November 1994.

[96] Trent Jaeger, Reiner Sailer, and Umesh Shankar. PRIMA: policy-reduced integrity measurement architecture. In *SACMAT '06: Proceedings of the 11th ACM Symposium on Access Control Models and Technologies*, pages 19–28. ACM, 2006.

[97] JAX-WS Reference Implementation Project. https://jax-ws.dev.java.net/, 2009.

[98] S. Jiang, S. Smith, and K. Minami. Securing Web Servers against Insider Attack. In *ACSAC '01: Proceedings of the 17th Annual Computer Security Applications Conference*, page 265, Washington, DC, USA, 2001. IEEE Computer Society.

[99] Willem Jonker and Milan Petkovic, editors. *SDM 2009: Proceedings of the 6th VLDB Workshop on Secure Data Management*, **5776** of *Lecture Notes in Computer Science*. Springer, August 2009.

[100] J.R.Fisher. Prolog Tutorial. http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html. Retrieved in August 2010.

[101] Trusted Computing for the Java(tm) Platform. http://trustedjava.sourceforge.net/, 2010.

[102] Chris Karlof, Naveen Sastry, and David Wagner. Cryptographic Voting Protocols: A Systems Perspective. In *Proceedings of the 14th USENIX Security Symposium*, pages 33–50. USENIX, 2005.

[103] Bernhard Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of the 16th USENIX Security Symposium*, pages 229–237. USENIX, 2007.

[104] Taghi M. Khoshgoftaar, Edward B. Allen, Kalai S. Kalaichelvan, and Nishith Goel. Early Quality Prediction: A Case Study in Telecommunications. *IEEE Software*, **13**(1):65–71, 1996.

[105] Chongkyung Kil, Emre Can Sezer, Ahmed Azab, Peng Ning, and Xiaolan Zhang. Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence. In *DSN '09: Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Lisbon, Portugal, June 2009. IEEE Press.

[106] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220, New York, NY, USA, 2009. ACM.

[107] Jacek Kopecký, Tomas Vitvar, Carine Bournez, and Joel Farrell. SAWSDL: Semantic Annotations for WSDL and XML Schema. *IEEE Internet Computing*, **11**(6):60–67, 2007.

[108] Dexter Kozen. Language-Based Security. Technical Report 1813/7405, Department of Computer Science, Cornell University, June 1999.

[109] Brian Krebs. Payment Processor Breach May Be Largest Ever. *The Washington Post Website*, 20th January 2009. http://voices.washingtonpost.com/securityfix/2009/01/payment_processor_breach_may_b.html?hpid=topnews.

[110] Ulrich Kühn, Marcel Selhorst, and Christian 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*, pages 50–57, New York, NY, USA, November 2007. ACM.

[111] David Kyle and José Carlos Brustoloni. Uclinux: a linux security module for trusted-computing-based usage controls enforcement. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable Trusted Computing*, pages 63–70, New York, NY, USA, 2007. ACM.

[112] G. Leavens and Y. Cheon. Design by Contract with JML. ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf, September 2006.

[113] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *NSDI '09: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2009.

[114] Steve Lipner. The Trustworthy Computing Security Development Lifecycle. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

[115] Yin-Soon Loh, Wei-Chuen Yau, Chien-Thang Wong, and Wai-Chuen Ho. Design and Implementation of an XML Firewall. In *CIS '06: Proceedings of the International Conference on Computational Intelligence and Security*, **2**, pages 1147–1150, November 2006.

[116] Hans Löhr, Hari Govind V. Ramasamy, Ahmad-Reza Sadeghi, Stefan Schulz, Matthias Schunter, and Christian Stüble. Enhancing Grid Security Using Trusted Virtualization. In Bin Xiao, Laurence T. Yang, Jianhua Ma, Christian Muller-Schloer, and Yu Hua, editors, *ATC '07: 4th International Conference on Autonomic and Trusted Computing*, **4610** of *Lecture Notes In Computer Science*, pages 372–384. Springer, July 2007.

[117] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John 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*. NIST, October 1998.

[118] Peter A. Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward McDonell. Linux kernel integrity measurement using contextual inspection. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable Trusted Computing*, pages 21–29, New York, NY, USA, 2007. ACM.

[119] Gavin Lowe, Philippa Broadfoot, Christopher Dilloway, and Mei Lin Hui. *Casper: A Compiler for the Analysis of Security Protocols*. Oxford University Computing Laboratory, Wolfson Building, Parks Road Oxford, OX1 3QD, UK, 1.12 edition, September 2009.

[120] Paul Jay Lucas. *An Object-oriented Language System For Implementing Concurrent, Hierarchical, Finite State Machines*. Master's thesis, University of Illinois at Urbana-Champaign, 1993. http://homepage.mac.com/pauljlucas/resume/pjl-chsm-thesis.pdf.

[121] John Lyle. Trustable Remote Verification of Web Services. In Chen et al. [37], pages 153–168.

[122] John Lyle and Andrew Martin. On the Feasibility of Remote Attestation for Web Services. In *SecureCom 09: Proceedings of the International Symposium on Secure Computing*, **3**, pages 283–288, Los Alamitos, CA, USA, September 2009. IEEE Computer Society.

[123] John Lyle and Andrew Martin. Engineering attestable services (short paper). In Alessandro Acquisti, Sean W. Smith, and Ahmad-Reza Sadeghi, editors, *TRUST 2010: Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, Lecture Notes in Computer Science, pages 257–264. Springer, June 2010.

[124] John Lyle and Andrew Martin. Trusted Computing and Provenance: Better Together. In *TaPP '10: Proceedings of the 2nd Workshop on the Theory and Practice of Provenance*. USENIX, 2010.

[125] T. S. E. Maibaum. Challenges in Software Certification. In Michael G. Hinchey Michael Butler and Maria M. Larrondo-Petrie, editors, *ICFEM '07: Proceedings of the 9th International Conference on Formal Engineering Methods*, **4789** of *Lecture Notes in Computer Science*, pages 4–18. Springer, 2007.

[126] John Marchesini, Sean Smith, Omen Wild, and Rich MacDonald. Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear. Technical Report TR2003-476, Department of Computer Science/Dartmouth PKI Lab, Hanover, New Hampshire USA, December 2003. http://www.cs.dartmouth.edu/~sws/pubs/TR2003-476.pdf.

[127] Andrew Martin and Po-Wah Yau. Grid security: Next steps. *Information Security Technical Report*, **12**(3):113 – 122, 2007.

[128] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic Markup for Web Services, W3C Member Submission. http://www.w3.org/Submission/OWL-S/, November 2004.

[129] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *S&P '10: Proceedings of IEEE Symposium on Security and Privacy (Oakland 2010)*, pages 143–158. IEEE, May 2010.

[130] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Arvind Seshadri. Minimal TCB Code Execution. In *S&P '07: Proceedings of the IEEE Symposium on Security and Privacy*, pages 267–272. IEEE, May 2007.

[131] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for TCB minimization. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 315–328, New York, NY, USA, 2008. ACM.

[132] H. McGhan and M. O'Connor. PicoJava: a direct execution engine for Java bytecode. *Computer*, **31**(10):22–30, October 1998.

[133] D. Harrison McKnight and Norman L. Chervany. The Meanings Of Trust. Technical Report wp96-04, Carlson School of Management, University of Minnesota, November 1996. http://www.misrc.umn.edu/wpaper/wp96-04.htm.

[134] Ce Meng, Yeping He, and Qian Zhang. Remote Attestation for Custom-built Software. In *NSWCTC '09: Proceedings of the International Conference on Networks Security, Wireless Communications and Trusted Computing*, **2**, pages 374–377, Wuhan, Hubei, April 2009. IEEE.

[135] Rebecca T. Mercuri. The HIPAA-potamus in Health Care Data Security. *Communications of the ACM*, **47**(7):25–28, July 2004.

[136] Bertrand Meyer. *Object-Oriented Software Construction*, chapter Design by Contract: Building Reliable Software, pages 331–341. Prentice Hall, 1997.

[137] Simon Miles, Paul Groth, Steve Munroe, and Luc Moreau. PrIMe: A Methodology for Developing Provenance-Aware Applications. Technical Report 13215, University of Southampton Schoold of Electronics and Computer Science, June 2009. http://eprints.ecs.soton.ac.uk/13215/.

[138] Mitre Organisation. CVE: Common Vulnerabilites and Exposures Website. http://cve.mitre.org/.

[139] Luc Moreau, Paul Groth, Simon Miles, Javier Vazquez-Salceda, John Ibbotson, Sheng Jiang, Steve Munroe, Omer Rana, Andreas Schreiber, Victor Tan, and Laszlo Varga. The provenance of electronic data. *Communications of the ACM*, **51**(4):52–58, April 2008.

[140] Luc Moreau, Bertram Ludscher, Ilkay Altintas, Roger S. Barga, Shawn Bowers, Steven Callahan, George Chin JR., Ben Clifford, Shirley Cohen, Sarah Cohen-Boulakia, Susan Davidson, Ewa Deelman, Luciano Digiampietri, Ian Foster, Juliana Freire, James Frew, Joe Futrelle, Tara Gibson, Yolanda Gil, Carole Goble, Jennifer Golbeck, Paul Groth, David A. Holland, Sheng Jiang, Jihie Kim, David Koop, Ales Krenek, Timothy McPhillips, Gaurang Mehta, Simon Miles, Dominic Metzger, Steve Munroe, Jim Myers, Beth Plale, Norbert Podhorszki, Varun Ratnakar, Emanuele Santos, Carlos Scheidegger, Karen Schuchardt, Margo Seltzer, Yogesh L. Simmhan, Claudio Silva, Peter Slaughter, Eric Stephan, Robert Stevens, Daniele Turi, Huy Vo, Mike Wilde, Jun Zhao, and Yong Zhao. Special Issue: The First Provenance Challenge. *Concurrency and Computation: Practice and Experience*, **20**(5):409–418, 2008.

[141] Seiji Munetoh. Practical Integrity Measurement and Remote Verification for Linux Platform. In *WATC '06: Proceedings of The Second Workshop on Advances in Trusted Computing*, 2006.

[142] Seiji Munetoh, Megumi Nakamura, Sachiko Yoshihama, and Michiharu Kudo. Integrity Management Infrastructure for Trusted Computing. *IEICE Transactions on Information and Systems*, **E91-D**(5):1242–1251, 2008.

[143] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *ATEC '06: Proceedings of the USENIX Annual Technical Conference*, pages 4–4, Berkeley, CA, USA, 2006. USENIX.

[144] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo I. Seltzer. Making a Cloud Provenance-Aware. In Cheney [38].

[145] Aarthi Nagarajan, Vijay Varadharajan, and Michael Hitchens. Trust management for trusted computing platforms in web services. In *STC '07: Proceedings of the 2007 ACM workshop on Scalable Trusted Computing*, pages 58–62, New York, NY, USA, 2007. ACM.

[146] Cornelius Namiluko. *Trusted Infrastructure for the Campus Grid*. Master's thesis, Oxford University Computing Laboratory, Wolfson College, Oxford, September 2008. http://www.comlab.ox.ac.uk/files/2648/CorneliusNamilukoThesis.pdf.

[147] Mohammad Nauman, Masoom Alam, Xinwen Zhang, and Tamleek Ali. Remote Attestation of Attribute Updates and Information Flows in a UCON System. In Chen et al. [37], pages 63–80.

[148] Mohammad Nauman, Sohail Khan, Xinwen Zhang, and Jean-Pierre Seifert. Beyond Kernel-level Integrity Measurement: Enabling Remote Attestation for the Android Platform. In Alessandro Acquisti, Sean W. Smith, and Ahmad-Reza Sadeghi, editors, *Trust '10: Proceedings of the 3rd International Conference on Trust and Trustworthy Systems*, Lecture Notes in Computer Science, Berlin, Germany, June 2010. Springer.

[149] George Necula. Proof-Carrying Code. http://raw.cs.berkeley.edu/pcc.html, July 2002.

[150] Qun Ni, Shouhuai Xu, Elisa Bertino, Ravi S. Sandhu, and Weili Han. An Access Control Language for a General Provenance Model. In Jonker and Petkovic [99], pages 68–88.

[151] Andreas Niederl and Martin Pirker. Build Guide for Bootstrapping a Reduced Trusted Java Compartment. http://trustedjava.sourceforge.net/index.php?item=pca/compartment, March 2009.

[152] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *CCGRID '09: Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[153] OASIS. Web Services Security: SOAP Message Security 1.1. http://docs.oasis-open.org/wss/v1.1/, 2004.

[154] OASIS. *WS-Trust Specification*, 1.4 edition, February 2009. http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/os/ws-trust-1.4-spec-os.html.

[155] Open Platform Trust Service Website. http://sourceforge.jp/projects/openpts/, 2009.

[156] Michael P. Papazoglou and Jean-jacques Dubray. A Survey of Web Service Technologies. Technical Report DIT-04-058, Informatica e Telecomunicazioni, University of Trento, June 2004. http://eprints.biblio.unitn.it/archive/00000586/.

[157] Nathanael Paul and Andrew S. Tanenbaum. Trustworthy Voting: From Machine to System. *Computer*, **42**(5):23–29, May 2009.

[158] Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman, and Jean-Louis Lanet. Enforcing High-Level Security Properties for Applets. Technical Report 5061, Institut National de Recherche en Informatique et en Automatique, December 2004. http://hal.inria.fr/docs/00/07/15/23/PDF/RR-5061.pdf.

[159] The PCI Data Security Standard (PCI DSS). https://www.pcisecuritystandards.org, October 2008.

[160] Siani Pearson, Boris Balacheff, and Liqun Chen. *Trusted Computing Platforms: TCPA Technology in Context*, chapter 1, page 41. HP Professional Series. Prentice Hall PTR, 1 edition, July 2002.

[161] Eduardo Pelegri-Llopart, Yutaka Yoshida, and Alexis Moussine-Pouchkine. Delivering a Java EE Application Server. https://glassfish.dev.java.net/faq/v2/GlassFishOverview.pdf, June 2007.

[162] A. Perrig, S. Smith, D. Song, and J.D. Tygar. SAM: a flexible and secure auction architecture using trusted hardware. In *Proceedings 15th International Parallel and Distributed Processing Symposium*, pages 1764–1773. IEEE, April 2001.

[163] Ahmad-Reza Sadeghi Peter Lipp and Klaus-Michael Koch, editors. *Trust '08: Proceedings of the First International Conference on Trusted Computing and Trust in Information Technologies*, **4968/2008** of *Lecture Notes in Computer Science*, Villach, Austria, March 2008. Springer.

[164] Charles Petrie. Practical Web Services. *IEEE Internet Computing*, **13**(6):93–96, Nov.-Dec. 2009.

[165] Corin Pitcher and James Riely. Dynamic Policy Discovery with Remote Attestation. In *FoSSaCS '06: Proceedings of the 9th International Conference on Foundations of Software Science and Computation Structures*, **3921** of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2006.

[166] Jonathan Poritz, Matthias Schunter, Els Van Herreweghen, and Michael Waidner. Property Attestation - Scalable and Privacy-friendly Security Assessment of Peer Computers. Technical Report RZ 3548 (99559), IBM Research, IBM Zurich Research Laboratory, Zurich, Switzerland, October 2004.

[167] Jonathan 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*, pages 1855–1859, New York, NY, USA, 2006. ACM.

[168] D. J. Power, E. A. Politou, M. A. Slaymaker, and A. C. Simpson. Securing web services for deployment in health grids. *Future Generation Computer Systems*, **22**(5):547–570, 2006.

[169] G. J. Proudler. *Trusted Computing*, chapter Concepts Of Trusted Computing, pages 11–13. The Institution of Engineering and Technology, 2005.

[170] H. Rajan and M. Hosamani. Tisa: Toward Trustworthy Services in a Service-Oriented Architecture. *IEEE Transactions on Services Computing*, **1**(4):201–213, Oct.-Dec. 2008.

[171] Christine F. Reilly and Jeffrey F. Naughton. Transparently Gathering Provenance with Provenance Aware Condor. In Cheney [38].

[172] Frédéric Rioux and Patrice Chalin. Improving the Quality of Web-based Enterprise Applications with Extended Static Checking: A Case Study. *Electronic Notes in Theoretical Computer Science*, **157**(2):119–132, 2006.

[173] Gordon Thomas Rohrmair. *Using CSP to Verify Security-Critical Applications*. PhD thesis, Oxford University Computing Laboratory, Hilary Term 2005.

[174] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[175] K. Rustan and M. Leino. Extended Static Checking: A Ten-Year Perspective. In Reinhard Wilhelm, editor, *Proceedings of Informatics: 10 Years Back, 10 Years Ahead*, **2000** of *Lecture Notes in Computer Science*, pages 157–175. Springer, 2001.

[176] P. Y. A. Ryan and T. Peacock. Pret a Voter: a System Perspective. Technical Report CS-TR No 929, School of Computing Science, Newcastle University, September 2005.

[177] A.C. Sabelfeld, A.; Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, **21**(1):5–19, January 2003.

[178] Ahmad-Reza Sadeghi and Christian Stüble. Property-based Attestation for Computing Platforms: Caring About Properties, Not Mechanisms. In *NSPW '04: Proceedings of the 2004 Workshop on New Security Paradigms*, pages 67–77, New York, NY, USA, 2004. ACM.

[179] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 308–317, New York, NY, USA, 2004. ACM.

[180] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th USENIX Security Symposium*, pages 223–238. USENIX, 2004.

[181] Ravi Sandhu and Xinwen Zhang. Peer-to-peer access control architecture using trusted computing technology. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access Control Models and Technologies*, pages 147–158, New York, NY, USA, 2005. ACM Press.

[182] Daniel Sandler and Dan S. Wallach. Casting votes in the auditorium. In *EVT'07: Proceedings of the USENIX / Accurate Workshop on Electronic Voting Technology*, pages 4–4, Berkeley, CA, USA, 2007. USENIX.

[183] Daniel R. Sandler, Kyle Derr, and Dan S. Wallach. VoteBox: a tamper-evident, verifiable electronic voting system. In *EVT'08: Proceedings of the USENIX / Accurate Workshop on Electronic Voting Technology*. USENIX, July 2008.

[184] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards Trusted Cloud Computing. In *HotCloud '09: Proceedings of the Workshop on Hot Topics In Cloud Computing*. USENIX, June 2009.

[185] Can Sar and Pei Cao. The lineage file system. Online at http://theory.stanford.edu/~cao/lineage. Last modified May 2005.

[186] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *STC '06: Proceedings of the first ACM workshop on Scalable Trusted Computing*, pages 27–42, New York, NY, USA, 2006. ACM.

[187] Beata Sarna-Starosta, R. E. K. Stirewalt, and Laura K. Dillon. Contracts and Middleware for Safe SOA Applications. In *SDSOA '07: Proceedings of the International Workshop on Systems Development in SOA Environments*, page 5, Washington, DC, USA, 2007. IEEE Computer Society.

[188] Dries Schellekens, Brecht Wyseur, and Bart Preneel. Remote Attestation on Legacy Operating Systems With Trusted Platform Modules. *Electronic Notes in Theoretical Computer Science*, **197**(1):59–72, 2008.

[189] Natasha Sharygina and Daniel Kröning. *Test and Analysis of Web Services*, chapter Model Checking with Abstraction for Web Services, pages 121–145. Computer Science. Springer, 2007.

[190] Mary Shaw. Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, page 181, Washington, DC, USA, 1996. IEEE Computer Society.

[191] Kanna Shimizu, Stefan Nusser, Wilfred Plouffe, Vladimir Zbarsky, Masaharu Sakamoto, and Masana Murase. Cell Broadband Engine: processor security architecture and digital content protection. In *MCPS '06: Proceedings of the 4th ACM International Workshop on Contents Protection and Security*, pages 13–18, New York, NY, USA, 2006. ACM.

[192] R. Shirey. Internet Security Glossary, Version 2. RFC 4949 (Informational) http://www.ietf.org/rfc/rfc4949.txt, August 2007. Obsoletes: 2828.

[193] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *ACM SIGMOD Record Newsletter*, **34**(3):31–36, September 2005.

[194] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. In *EuroSys '06: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 161–174, New York, NY, USA, 2006. ACM.

[195] Munindar P. Singh and Michael N. Huhns. *Service-Oriented Computing*, chapter Basic Standards for Web Services, pages 19–47. Wiley, November 2004.

[196] Munindar P. Singh and Michael N. Huhns. *Service-Oriented Computing*, chapter Principles of Service-Oriented Computing, pages 74–75. Wiley, November 2004.

[197] Anoop Singhal, Theodore Winograd, and Karen Scarfone. Guide to Secure Web Services. http://csrc.nist.gov/publications/nistpubs/800-95/SP800-95.pdf, August 2007.

[198] L. St. Clair, J. Schiffman, T. Jaeger, and P. McDaniel. Establishing and Sustaining System Integrity via Root of Trust Installation. In *ACSAC '07: Proceedings of the Twenty-Third Annual Computer Security Applications Conference*, pages 19–29. IEEE, December 2007.

[199] David Stainforth, Andrew Martin, Andrew Simpson, Carl Christensen, Jamie Kettleborough, Tolu Aina, and Myles Allen. Security principles for public-resource modeling research. In *WETICE '04: Proceedings of the 13th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 319–324, Washington, DC, USA, 2004. IEEE Computer Society.

[200] Frederic Stumpf, Andreas Fuchs, Stefan Katzenbeisser, and Claudia Eckert. Improving the scalability of platform attestation. In *STC '08: Proceedings of the 3rd ACM workshop on Scalable Trusted Computing*, pages 1–10, New York, NY, USA, 2008. ACM.

[201] Frederic Stumpf, Omid Tafreschi, Patrick Röder, and Claudia Eckert. A Robust Integrity Reporting Protocol for Remote Attestation. In *WATC '06: Proceedings of The Second Workshop on Advances in Trusted Computing*, 2006.

[202] Masoom Alam Tamleek Ali Tanveer and Muhammad Nauman. Scalable Remote Attestation with Privacy Protection. In Liqun Chen and Moti Yung, editors, *INTRUST '09: Proceedings of the First International Conference on Trusted Systems*, **6163** of *Lecture Notes In Computer Science*. Springer, December 2009.

[203] Victor Tan, Paul T. Groth, Simon Miles, Sheng Jiang, Steve Munroe, Sofia Tsasakou, and Luc Moreau. Security Issues in a SOA-Based Provenance System. In Luc Moreau and Ian Foster, editors, *IPAW'06: Proceedings of the International Provenance and Annotation Workshop*, **4145** of *Lecture Notes in Computer Science*, pages 203–211, Chicago, IL, USA, May 2006. Springer.

[204] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can We Make Operating Systems Reliable and Secure? *IEEE Computer*, **39**(5):44–51, May 2006.

[205] Christopher Tarnovsky. Deconstructing a 'secure' processor. http://www.blackhat.com/presentations/bh-dc-10/Tarnovsky_Chris/BlackHat-DC-2010-Tarnovsky-DASP-slides.pdf, February 2010. Presented at BlackHat DC 2010.

[206] The OpenTC Project. Proof of Concept Prototype. http://www.opentc.net/index.php?option=com_content&task=view&id=45&Itemid=63, 2007.

[207] The Trusted Computing Group. TCG Infrastructure Working Group Architecture Part II - Integrity Management, November 2006.

[208] The Trusted Computing Group. Tcg infrastructure working group platform trust services interface specification (if-pts). http://www.trustedcomputinggroup.org/resources/infrastructure_work_group_platform_trust_services_interface_specification_ifpts_version_10, November 2006. Version 1.0.

[209] The Trusted Computing Group. TPM Main Specification, Part 3: Commands. http://www.trustedcomputinggroup.org/resources/tpm_main_specification, October 2006.

[210] The Trusted Computing Group. Frequently Asked Questions. http://www.trustedcomputinggroup.org/faq/, October 2007.

[211] The Trusted Computing Group. TCG Software Stack (TSS) Specification Version 1.2. http://www.trustedcomputinggroup.org/resources/tcg_software_stack_tss_specification, March 2007.

[212] The Trusted Computing Group. *TPM Main Specification: Part 1 Design Principles*, revision 103 edition, July 2007.

[213] The Trusted Computing Group. Trusted Network Connect - Frequently Asked Questions. http://www.trustedcomputinggroup.org/developers/trusted_network_connect/faq, September 2009. Accessed September 2009.

[214] The Trusted Computing Group. Website. http://www.trustedcomputinggroup.org/, 2009.

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

[216] R. Thibadeau. Trusted Computing for Disk Drives and Other Peripherals. *IEEE Security & Privacy*, **4**(5):26–33, Sept.-Oct. 2006.

[217] Ken Thompson. Reflections on trusting trust. *ACM Turing Award Lectures*, 1983.

[218] Ronald Toegl and Siegfried Podesser. A Software Architecture for Introducing Trust in Java-based Clouds. In *Presented at The Workshop on Trust in the Cloud*, Berlin, Germany, June 2010. .

[219] TrouSerS: the open-source TCG software stack. http://trousers.sourceforge.net/, 2008.

[220] Trusted Computing Group. TCG Schema. http://www.trustedcomputinggroup.org/resources/infrastructure_work_group_reference_manifest_rm_schema_specification_version_10, November 2006.

[221] Trusted Computing Group. Summary Of Features Under Consideration For The Next Generation Of TPM. http://www.trustedcomputinggroup.org/resources/summary_of_features_under_consideration_for_the_next_generation_of_tpm, 2009.

[222] Trusted Computing Group. TCG Storage Architecture Core Specification. http://www.trustedcomputinggroup.org/resources/tcg_storage_architecture_core_specification, April 2009.

[223] W.T. Tsai, X. Wei, Y. Chen, B. Xiao, R. Paul, and H. Huang. Developing and assuring trustworthy Web services. In *ISADS '05: Proceedings of the Internation Symposium on Autonomous Decentralized Systems*, pages 43–50. IEEE, April 2005.

[224] Ubuntu Packages Search Website. http://packages.ubuntu.com/, 2009.

[225] Ülle Madise and Tarvi Martens. E-voting in Estonia 2005. The first practice of country-wide binding Internet voting in the world. In Robert Krimmer, editor, *Proceedings of the 2nd International Workshop on Electronic Voting*, **86** of *GI Lecture Notes in informatics*, pages 15–26, Castle Hofen, Bregenz, Austria, August 2006.

[226] United States House of Representatives. Health Insurance Portability and Accountability Act. In *Congressional Reports*, number H. Rept. 104-736 in Congressional Committee Materials. U.S. Government Printing Office, July 1996.

[227] U.S. Department of Defense. *Trusted Computer System Evaluation Criteria*, dod-5200.28-std edition, December 1985. http://www.dynamoo.com/orange/fulltext.htm.

[228] Amin Vahdat and Thomas Anderson. Transparent result caching. In *ATEC '98: Proceedings of the USENIX Annual Technical Conference*, pages 3–3, Berkeley, CA, USA, 1998. USENIX.

[229] J. Vázquez-Salceda, S. Alvarez, T. Kifor, L. Z. Varga, S. Miles, L. Moreau, and S. Willmott. *Agent Technology and E-Health*, chapter EU PROVENANCE Project: An Open Provenance Architecture for Distributed Applications. Whitestein Series in Software Agent Technologies and Autonomic Computing. Birkhäuser Verlag AG, Switzerland, December 2007.

[230] Willem Visser and Peter C. Mehlitz. Model Checking Programs with Java PathFinder. In *SPIN '05: Proceedings of the 12th International SPIN Workshop on Model Checking Software*, **3639** of *Lecture Notes in Computer Science*, page 27. Springer, August 2005.

[231] Dongbo Wang and Ai min Wang. Trust Maintenance Toward Virtual Computing Environment in the Grid Service. In Yanchun Zhang, Ge Yu, Elisa Bertino, and Guandong Xu, editors, *APWeb '08: Proceedings of the 10th Asia-Pacific Web Conference*, **4976** of *Lecture Notes in Computer Science*, pages 166–177. Springer, April 2008.

[232] Hua Wang, Yao Guo, and Xiangqun Chen. SAConf: Semantic Attestation of Software Configurations. In Guojun Wang Juan Gonzalez Nieto, Wolfgang Reif and Jadwiga Indulska, editors, *ATC '09: Proceedings of the 6th International Conference on Autonomic and Trusted Computing*, **5586** of *Lecture Notes in Computer Science*, pages 120–133, Berlin, Heidelberg, July 2009. Springer.

[233] Willis H. Ware. Security and privacy in computer systems. In *AFIPS '67: Proceedings of the Spring Joint Computer Conference*, pages 279–282, New York, NY, USA, April 1967. ACM.

[234] Yuji Watanabe, Sachiko Yoshihama, Takuya Mishina, Michiharu Kudo, and Hiroshi Maruyama. Bridging the Gap Between Inter-communication Boundary and Internal Trusted Components. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *ESORICS '06: Proceedings of the 11th European Symposium on Research in Computer Security*, **4189** of *Lecture Notes in Computer Science*, pages 65–80. Springer, September 2006.

[235] Paul A. Watters. *Web Services in Finance*. APRESS Academic, December 2004. http://www.apress.com/9781590594353.

[236] Jinpeng Wei, L. Singaravelu, and C. Pu. A Secure Information Flow Architecture for Web Service Platforms. *IEEE Transactions on Services Computing*, **1**(2):75–87, April-June 2008.

[237] Bart De Win, Bart Vanhaute, and Bart De Decker. Security Through Aspect-Oriented Programming. In Bart De Decker, Frank Piessens, Jan Smits, and Els Van Herreweghen, editors, *Proceedings of the IFIP TC11 WG11.4 First Annual Working Conference on Network Security: Advances in Network and Distributed Systems Security*, **206** of *IFIP Conference Proceedings*, pages 125–138. Kluwer, 2001.

[238] Ka-Ping Yee. User Interaction Design for Secure Systems. In Robert H. Deng, Sihan Qing, Feng Bao, and Jianying Zhou, editors, *ICICS '02: Proceedings of the 4th International Conference on Information and Communications Security*, **2513** of *Lecture Notes in Computer Science*, pages 278–290. Springer, 2002.

[239] S. Yoshihama, T. Ebringer, M. Nakamura, S. Munetoh, and H. Maruyama. WS-attestation: efficient and fine-grained remote attestation on Web services. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services*, pages 743–750. IEEE, July 2005.

[240] M. Zabel, T.B. Preusser, P. Reichel, and R.G. Spallek. Secure, Real-Time and Multi-Threaded General-Purpose Embedded Java Microarchitecture. In *DSD '07: Proceedings*

of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, pages 59–62. IEEE, August 2007.

[241] A. Zakinthinos and E.S. Lee. A general theory of security properties. In *S&P '97: Proceedings of the IEEE Symposium on Security and Privacy*, pages 94 –102. IEEE, 4-7 1997.

[242] Huanguo Zhang and Fan Wang. A behavior-based remote trust attestation model. *Wuhan University Journal of Natural Sciences*, **11**(6):1819–1822, May 2006.

[243] Jing Zhang, Adriane Chapman, and Kristen LeFevre. Do You Know Where Your Data's Been? - Tamper-Evident Database Provenance. In Jonker and Petkovic [99], pages 17–32.

Glossary

A2M Attested Append-only Memory. A secure history or log of events, page 43.

AIK Attestation Identity Key. An anonymised key generated by a Trusted Platform Module and used to sign TPM Quotes, page 19.

Ant A language and tool for writing compilation scripts, page 123.

Assurance The process of building evidence to show that something is trustworthy, page 6.

Attestation The process of reporting (or *attesting*) the configuration of a computing platform, page 19.

Authenticated boot When the boot process of a platform follows 'measure-before-load,' and every component is recorded in platform configuration registers, page 19.

Ballot box Part of an electronic voting system, designed to accept and store votes as they are cast, page 177.

Booster A domain specific language and code generator. The Booster compiler can take the formal definition of an information system and automatically generate a complete object database implementing it, page 29.

CA Certificate Authority.

CertifyInfo The credential for a TPM-bound key, showing that the private half of it is held in the TPM, page 26.

Chain of trust An ordered list of components on a system that are relied upon for trustworthy behaviour, including all hardware and software. Assurance of each link in the chain is dependent on the trustworthiness of every earlier component, page 18.

Composite Services Services which themselves use other services to complete their tasks, page 81.

CSP Communicating Sequential Processes, page 97.

D4V Design for Verification. Software engineering approach that requires code to be written so that it is amenable to machine verification, page 30.

DAA Direct Anonymous Attestation. Attestation without the use of an AIK, page 19.

DbC Design by Contract. The software engineering approach that requires all modules to have explicit contracts and responsibilities, page 27.

DRTM Dynamic Root of Trust for Measurement. The beginning of a new trust-chain that is initiated after platform boot via a special CPU instruction, page 21.

ESC Extended Static Checking code analysis technique, page 27.

ESC/Java Extended Static Checking for Java and JML, page 27.

Event Reporting Using platform configuration registers to record application-level state changes and events, page 50.

Glassfish The Glassfish Application Server. Web service middleware designed to host Java web services, page 63.

HMAC Hash-based Message Authentication Code, page 25.

IMA Integrity Measurement Architecture. A Linux Security Module that provides integrity measurement for the kernel. IMA-enabled systems will measure kernel modules, applications, and can be configured to measure arbitrary files, page 42.

IML Integrity Measurement Log. A record of all integrity measurements (hashes) recorded in platform configuration registers, page 19.

Introspection On platforms supporting virtual machines, a 'guest' VM instance is monitored by another, usually for intrusion detection. The monitor may inspect memory and system state in considerable detail, page 44.

JML Java Modelling Language. Annotations for Java specifying, amongst other constraints, pre- and post-conditions for methods, page 27.

JRE Java Runtime Environment.

JVM Java Virtual Machine.

Middleware Problem The problem that service middleware is large and complex, but placed in the trusted computing base, making it a target for attack, page 45.

MLE Measured Launch Environment, page 21.

Monotonic Counter Simple integer counters within the TPM which are incremented through TPM commands, and can never be decremented, page 22.

Nonce A freshly-made random integer, used to establish timeliness, page 25.

PBA Property-based Attestation. A layer of indirection between attestation and security statements, so that general security properties are attested rather than binary hashes, page 41.

PCA Privacy CA. Certificate Authority responsible for certifying that an Attestation Identity Key comes from a real TPM, page 26.

PCC Proof-Carrying Code, page 29.

PCR Platform Configuration Registers. Registers in the Trusted Platform Module that can only be modified through the 'extend' operation, and platform reboot, page 18.

PRIMA A version of the IMA system, integrating SELinux policies to avoid the need for measuring untrusted software, page 42.

Provenance Provenance or lineage generally refers to information that 'helps determine the derivation history of a data product, starting from its original sources', page 4.

RIM Reference Integrity Measurements. Hashes of the binaries that represent and uniquely identify an application, page 19.

RMDB Reference Manifest Database. A database of RIMs, page 19.

Root of Trust The first component in a trust chain, a trusted component which is relied upon to assess the trustworthiness of the rest of the platform, page 20.

RSA Rivest, Shamir and Adleman's algorithm for public-key encryption, page 18.

RTM Root of Trust for Measurement. Either static (STRM) or dynamic (DRTM), page 20.

RTR Root of Trust for Reporting, page 21.

RTS Root of Trust for Storage, page 21.

Semantic Gap Problem with attestation referring to the difference between reporting platform execution state and security state, page 35.

SHA-1 Secure Hashing Algorithm version 1, page 18.

SOA Service Oriented Architecture, page 15.

SOAP A web service messaging protocol, SOAP original stood for the 'Simple Object Access Protocol' [215], page 16.

SOC Service Oriented Computing, page 15.

SRA Semantic Remote Attestation, page 43.

SRTM Static Root of Trust for Measurement. The first component in the boot-chain, typically the first sector of the BIOS, page 20.

TCB Trusted computing base, page 8.

TCG Trusted Computing Group. Industry body responsible for defining standards on protocols and components such as the Trusted Platform Module, Trusted Network Connect, page 18.

TGA Trusted Grid Architecture, page 46.

Tick Counter TPM counter used for time keeping, page 23.

TNC Trusted Network Connect, page 23.

TPDMenu Trusted PDMenu. PDMenu is a menu-based shell used in place of standard Unix shells such as BASH. It limits the user to running commands pre-defined in its configuration file. The trusted version uses the TPM to measure commands as they are executed, page 108.

TPM Trusted Platform Module, page 18.

TPM Bind A TPM Command. Bind encrypts data to a specific TPM, so that only that TPM can decrypt it, page 22.

TPM Quote The information reported in an attestation. This is generated by the Trusted Platform Module and contains the values of the platform configuration registers. Signed by an Attestation Identity Key (AIK), page 19.

TPM Seal A TPM Command. Seal encrypts data to a specific TPM, so that only that TPM can decrypt it, page 22.

TRC Trustable remote compilation, as defined in this dissertation, page 127.

Trust An overloaded term, used in this dissertation to mean 'belief' or 'faith', page 6.

Trusted Computing Umbrella term for all technology and standards developed by the Trusted Computing Group, page 18.

Trustworthy Something is trustworthy if it *will* behave in a reliable, expected manner, page 6.

TRV Trustable remote verification, as defined in this dissertation, page 118.

TSS Trusted Software Stack. Software responsible for managing the Trusted Platform Module and providing an API for applications to use, page 22.

UCLinux A Linux Security Module designed to implement usage-controls, page 44.

VM Virtual Machine, page 23.

vTPM Virtual TPM. A software TPM provided to virtual machines in order to fully virtualize the real hardware platform [14] .

Whitelist A list of trustworthy software, used to validate a platform's integrity measurement logs. If the log contains an entry not on the list, the platform will not be considered trustworthy, page 37.

WSDL Web Service Description Language. Used to define web service interfaces, page 16.

Appendix A

A Trusted Ballot Box Service

It's not the people who vote that count, it's the people who count the votes

Joseph Stalin

This appendix describes the specification of a ballot box service for electronic voting, designed so that ballots are extended into PCRs when cast. This example has been implemented, and is used in the evaluation in Chapter 8.

A.1 Background

Electronic voting, for the purpose of this chapter, attempts to allow voters to cast a ballot without needing to physically go to a polling station and fill in a form. Instead, home users can vote using an online system. This has the advantage of making voting easier for those with disabilities and citizens living abroad, but has many security challenges.

The literature on electronic voting is extensive, and there are many implementations and voting models to consider. However, one of the primitive components of many systems is the ballot box (or bulletin board [176, 44]). These are eventually public (or semi-public) records of ballots cast. They often have few constraints, but are relied upon to maintain the *integrity* of the votes so that none are lost or modified. It has been pointed out in the past that such components are underspecified [102] and could be a target for attackers wishing to influence elections. Some researchers have already proposed the use of a secure coprocessor [98] to maintain confidentiality and attestation to verify that the correct software is being run on the voting machines [157]. The general election system can be seen in Figure A.1, this appendix is concerned with implementing stages 4, 5, 6, 7 and 9.

Requirements for confidentiality are not part of the problem for the ballot box, as in many proposed systems an earlier component implements the necessary cryptography. Indeed, in the Civitas system an earlier protocol involving the 'registration teller' performs this function [44]. Therefore in this appendix the primary concern is maintaining the *integrity* of ballots and

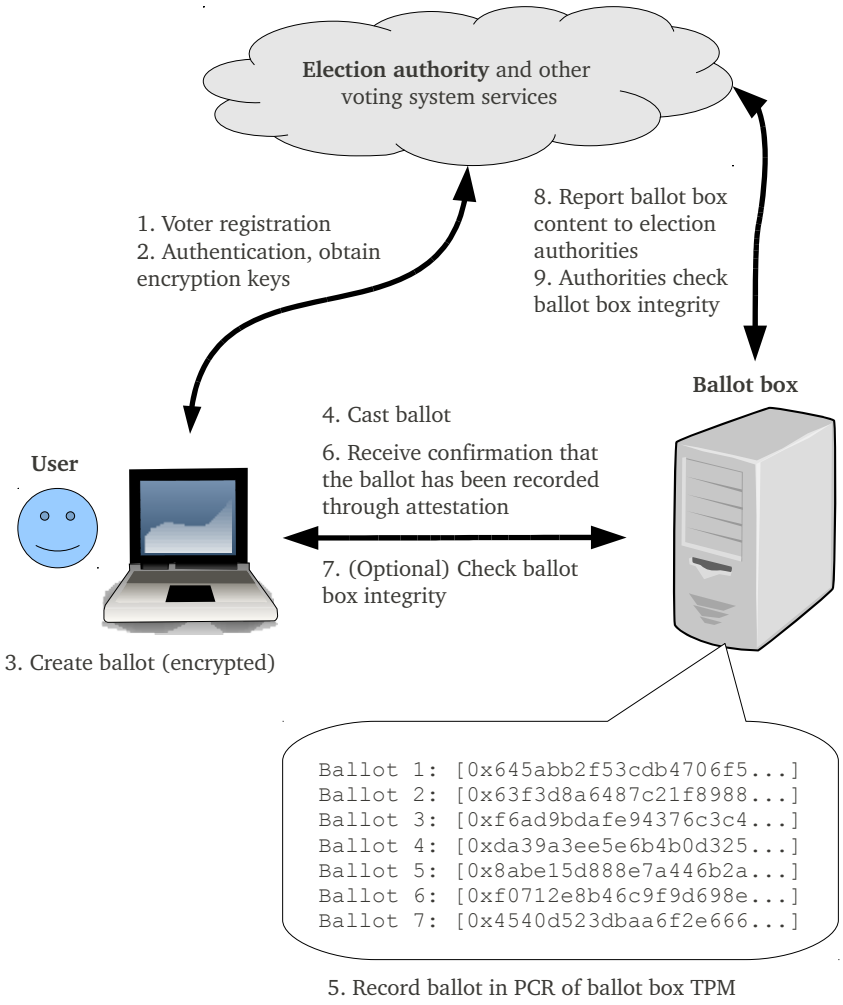

Figure A.1: Electronic voting system overview

making sure that any ballot cast is recorded and cannot be deleted at a later date. Any extra functionality required of the ballot box could be integrated with the approach defined in the rest of this appendix.

A.2 Requirements

The ballot box service will attempt to meet the following requirements:

1. Voters can confirm that their vote has been recorded.
2. Any user can request and check the content of the ballot box. This does not mean that any user can observe the content of the ballots themselves, as most electronic voting schemes use cryptography to maintain voter confidentiality. The ballot box must allow the *cipher text* of all ballots to be available.

3. The reported number of ballots must equal the number of submitted ballots.
4. The content of the ballot box must be equal to the accumulated content of submitted ballots.
5. Ballots must be reported by the ballot box in the order in which they were submitted.
6. Any ballot box that fails to record a vote cannot claim otherwise. This means that a ballot box cannot claim to have recorded a voter's input when it has not. Breaking this requirement would allow a rogue ballot box to discard many votes.
7. Any attack or compromise of the ballot box should not invalidate the ballots already cast.

A.3 Description and Operations

The system consists of a Ballot Box server, a Time Stamp Service and any number of voters. For the purpose of this specification, no consideration is given to the *content* of the ballots, only their presence on the system. This is all that is generally required by existing electronic voting systems.

The ballot box has four distinct operations, plus initialisation and shut down. These four are *ballot submission*, *list ballots* which returns a list of all recorded ballots, *timestamp*, for recording the current time in the log, and *checkpoint* which saves the current state of the platform. PCRs and TPM counters are used to provide integrity guarantees on each operation, so that votes cannot be lost after being successfully cast.

A.3.1 Ballot box initialisation

When the platform starts up, it must record its boot process in the first 8 PCRs (authenticated boot) and then start the Ballot Box (*BB*) application:

1. Platform boot (authenticated boot, using PCRs 0 to 7).
2. The operating system will call `extend(9,` $BinaryHash_{BB}$`)`.
3. Ballot box application loaded.
4. If a previous counter Id exists, set $counter_{Id}$ to this value. If not, `TPM_CreateCounter` is called to initialize a new counter with a new label. $counter_{Id}$ is set to this and stored in a configuration file.
5. Add to log: 'Ballot box started.'
6. `extend(10, 'Ballot box started')`.
7. `timestamp()`.
8. Start listening for messages.

Note that the counter label is effectively the label for this election. A TPM may only increment one counter per boot, but many can be stored in the TPM. As a result, the same platform can be restarted for use in different elections.

A.3.2 Timestamp

The purpose of the timestamp is to link the current PCR values and counter to a date and time. This is useful for demonstrating when messages were received, and for error recovery.

It is a three-step procedure, the first of which is to use the *Request Timestamp* protocol.

1. $TimeStamp_{TS}$ = `RequestTimestamp()`.
2. Add to log: $TimeStamp_{TS}$.
3. `extend(10,` $TimeStamp_{TS}$ `)`.

A.3.3 Request timestamp

The message exchange between the Ballot Box (*BB*) and Timestamp Service (*TS*). The Timestamp service must verify the credential in step one, and check the nonce in step three. The time stamp reply in step four asserts that, at time *CurrentTime*, the time stamp service had challenged and seen a correct reply from the Ballot Box, with the given PCR and counter values. Signatures made by the AIK in this protocol are implemented as TPM Quotes and recorded transport sessions.

$BB \rightarrow TS$	: $AIKCredential_{SK(PCA)}\{\mid AIK\text{-}PK(BB)_1 \mid\}$	[Request timestamp]
$TS \rightarrow BB$	: $nonce_{ts}$	[Attestation challenge]
$BB \rightarrow TS$	: $SIG_{AIK-SK(BB)_1}\{\mid pcr_{0-10}, counter, counter_{Id}, nonce_{ts} \mid\}$	[Attestation]
$TS \rightarrow BB$	: $SIG_{SK(TS)}\{\mid Time, pcr_{0-10}, counter, counter_{Id}, AIK\text{-}PK(BB)_1 \mid\}$	[Time stamp]

A.3.4 Submitting a ballot

1. Voters posts a message *M*.
2. `increment_counter(`*counter*`)`.
3. Add to log: *M*.
4. `extend( 10 , M )`.

The voter may wish to request a list of ballots after submitting their vote, to make sure it was recorded.

$Voter \rightarrow BB$	: $nonce_{user}$	[Challenge]
$BB \rightarrow Voter$	: $AIKCredential_{SK(PCA)}\{\mid AIK\text{-}PK(BB)_1 \mid\}$	[AIK Credential]
$BB \rightarrow Voter$	: Log	[Measurement Log]
$BB \rightarrow Voter$	: $SIG_{AIK-SK(BB)_1}\{\mid pcr_{0-10}, counter, counter_{Id}, nonce_{user} \mid\}$	[Credential]

A.3.5 Requesting a list of ballots

Voters (or other parties) may request a list of ballots. To obtain the current log, the following protocol is run:

This is a basic nonce-challenge protocol. The voter will need to verify the AIK credential and make sure that the log corresponds to the PCR values given. PCRs 0 through to 7 must be checked for a trustworthy boot process, PCR 9 needs to show a correct hash of the ballot box software, and the Log of PCR 10 should be analysed against the model of Section 8.2.6.

If the platform has been restarted during the election, the user may wish to receive older logs too. Logs and their credentials can be saved (see Section A.3.6) to disk, and then later returned as part of the second message.

A.3.6 Checkpointing

At various points, such as when powering down, the ballot box will want to save the current log of votes and store a credential asserting the correct sequence of events. Again, TPM Quotes and transport sessions are used for creating the credential (see Section 4).

1. `extend(10 ,'Ballot box checkpoint')`
2. `timestamp()`
3. Save log of pcr_{10} to a file
4. Create credential (TPM quote and counter attestation):
   $SIG_{AIK-SK(BB)_1}\{\mid pcr_0 - 7, PCR_10, counter, counter_{Id} \mid\}$,
   $AIKCredential_{SK(PCA)}\{\mid AIK - PK(BB)_1 \mid\}$

After a checkpoint, the current log should continue to be used, as PCR values cannot be reset. Any subsequent checkpoints (before platform reboot) will effectively replace this one.

Although the credential is not directly tied to the timestamp (it may happen at any time afterwards) any future messages will not be certified by this message (due to counter values) so its replay is not useful.

A.3.7 Ballot box shut-down

The ballot box can power down at any time, providing it performs a checkpoint first.

1. Ballot box stops accepting new messages.
2. Last messages are processed.

3. `checkpoint()`.

4. Close application and power down.

A.3.8 Example

Event	Counter	pcr_{0-7}	pcr_{10}	pcr_{11}
-	0	0x00	0x00	0x00
Boot	0	0x11	0x00	0x00
Board start	0	0x11	0x22	0x00
Board init	0	0x11	0x22	0+`begin`+ts_1
Add M1	1	0x11	0x22	0+`begin`+ts_1+M1
Add M2	2	0x11	0x22	0+`begin`+ts_1+M1+M2
Checkpoint	2	0x11	0x22	0+`begin`+ts_1+M1+M2+`checkpoint` +ts_2
			Log saved	
Add M3	3	0x11	0x22	0+`begin`+ts_1+M1+M2+`checkpoint` +ts_2+M3
Add M4	4	0x11	0x22	0+`begin`+ts_1+M1+M2+`checkpoint` +ts_2+M3+M4
Shutdown	4	0x11	0x22	0+`begin`+ts_1+M1+M2+`checkpoint` +ts_2+M3+M4+ts_3
			Log saved	
Boot	4	0x11	0x00	0x00
Board start	4	0x11	0x22	0x00
Board init	4	0x11	0x22	0+`begin`+ts_4
Add M5	5	0x11	0x22	0+`begin`+ts_5+M5
Add M6	6	0x11	0x22	0+`begin`+ts_5+M5+M6
Checkpoint	6	0x11	0x22	0+`begin`+ts_5+M5+M6+`checkpoint` +ts_6

A.3.9 Marking the beginning and end of an election

The beginning of an election period must be marked by the election authority. One way of doing so would be to submit a 'ballot' to the service which contained a signed message from the election authority. This could contain the election details, the ballot's counter ID, the counter start value, and a timestamp. Users could then expect to receive all ballots submitted after this first message. The end of an election could be called in a similar way. At the given shut-down time, the ballot-box could request a timestamp, and then stop accepting input. The election authority could send a special 'end-of-election' message, containing the counter value it can see at this time.

A.3.10 Verifying the ballot box

The ballot box produces a log every time it performs a `checkpoint` operation. A collection of these will describe the complete voting process. Verification can be performed by the user or an election authority to check that (a) only trustworthy software was used to record ballots (b)

every ballot recorded is also measured into the PCR and vice-versa. The following steps are required:

1. Verifier requests a list of all logs that were used during the term of the election
2. Verifier received logs including credentials from each time a `checkpoint` operation was invoked.
3. Verifier checks the credentials by:

   Checking the attestation identity key and PCA certificate

   Checking the signatures on the attestations against this key

   Checking that the attested PCR values match the log content

   Checking that there are no gaps in the reported logs of voting between checkpoints

   Checking that the reported ballot total matches the counter value

   Checking that each timestamp was from a trusted timestamp server

   Checking that all the software reported in every log of PCRs 0-10 are trusted

If any of these credentials is invalid, it could imply that the period of voting it covers was invalid. This might be because untrustworthy software was running, or that the logs were tampered with to add or remove votes. However, regardless of previous invalid logs, any log which does satisfy these checks can still be trusted. How to proceed from this stage is up to the election authority.

Appendix B

Example Scripts

B.1 Ant Compilation Script

The following Ant scripts demonstrate an integrated compilation process which can split the given service into both a WSDL web interface and RMI server, and wrap code compiled with JMLC, *and* provide a compilation receipt. Some of the constants and file names have been omitted for brevity, as well as some of the less exciting classpaths.

```
<?xml version="1.0"?>
<project name="timestamp-server" default="create-and-attest">
  <!-- Some file paths and settings -->
  <property name="interface.class.simplename"
      value="TimeStampJmlFrontEnd" />
  <property name="input.src" value="TrustworthyBB/src-ts" />
  <property name="jml.input.src" value="TrustworthyBB/src-java14-ts" />
  <property name="ts.output" value="ts-deploy" />
  <property name="output.dir" value="${ts.output}/tc" />
  <property name="log.file" value="${output.dir}/log.txt" />

  <!-- TPM and boot-specific properties -->
  <!-- Which PCR should i use during this compilation process? -->
  <property name="compile.pcr" value="12" />
  <!-- Which PCRs should i put in the build certificate? -->
  <property name="quote.pcrs"
      value="1,2,3,4,5,6,7,8,9,10,11,${compile.pcr}" />

  <!-- Where to store the quote -->
  <property name="quote.output.dir"
      value="${output.dir}/attestation" />
  <property name="quote.output.file"
      value="${quote.output.dir}/quote.xml" />

  <property name="trustablecompiler.home"
      value="TrustableCompiler/bin/" />

  <!-- TaskDefs -->
  <!-- using the TPM -->
```

```
<taskdef name="extend"      classname="....TpmExtendTask" .../>
<taskdef name="getkey"      classname="....TpmGetBoundKeyTask" .../>
<typedef name="pcr"         classname="....Pcr" .../>
<taskdef name="quote"       classname="....TpmQuoteTask" .../>
<!-- generating RMI/WS wrappers -->
<taskdef name="wrap-rmi"    classname="....RMIWrapTask" .../>
<taskdef name="wrap-ws2rmi" classname="....WS2RmiWrapTask" .../>

<!-- Compile the Java 6 code -->
<target name="compile-java" depends="clean">
  <javac srcdir="${input.src}"  ... />
</target>
<!-- Generate JML-annotated source with JMLC -->
<target name="jmlc-generate-source" depends="compile-java" >
  <jmlc source="true" ...   classpath="..." verbose="true">
    <fileset dir="${jml.input.src}" casesensitive="yes">
      <include name="**/*.java" />
    </fileset>
  </jmlc>
</target>

<!-- Compile the annotated source -->
<target name="compile-jml" depends="jmlc-generate-source" >
  <javac srcdir="${jml.output.src}" ... source="1.4" />
</target>

<!-- Create the RMI server interface -->
<target name="create-rmi-server" depends="compile-jml" >
  <!-- Wrap the interface class and generate an RMI service -->
  <wrap-rmi interfaceClass="${interface.class}"
      dest="${rmi.output.src}" />
  <!-- Compile the generated source -->
  <javac srcdir="${rmi.output.src}" ... />
</target>

<!-- JAR the server classes into two parts: main code and RMI wrapper -->
<target name="jar-rmi-server" depends="create-rmi-server" >
  <jar destfile="${rmi.server.jar.file}" >
     <fileset dir="${rmi.output.bin}" />
     <fileset dir="${base.output.bin}" />
     <fileset dir="${jml.output.bin}" />
  </jar>
  <jar destfile="${rmi.server.jar.wrapper.file}" >
     <fileset dir="${rmi.output.bin}" />
    <fileset dir="${base.output.bin}"
        includes="**/TimeStamp.class" />
  </jar>
  <!--  Extend the resulting RMI JAR files.  This is the
      important part of the compilation certificate.
      Record this action in a log file  -->
  <record name="${log.file}" append="true"  action="start" />
  <extend pcr="${compile.pcr}" >
    <fileset file="${rmi.server.jar.file}" />
    <fileset file="${rmi.server.jar.wrapper.file}" />
  </extend>
  <record name="${log.file}" append="true"  action="stop" />
```

```
</target>

<!-- Create a RMI to WS wrapper -->
<target name="create-ws2rmi-wrapper" depends="jar-rmi-server" >
  <!-- Generate wrapper source code -->
  <wrap-ws2rmi
      interfaceClass="${interface.class}"
      dest="${ws2rmi.output.src}"
      rmiRegistry="${rmi.server.hostname}" />
  <!-- Compile -->
  <javac srcdir="${ws2rmi.output.src}"  ... />
</target>

<!-- Create the WSDL to go with it -->
<target name="create-ws-server-wsdl" depends="create-ws2rmi-wrapper" >
  <wsgen
    destdir="${wsdl.output.bin}" genwsdl="true"
    sourcedestdir="${wsdl.output.src}">
      <classpath> ... </classpath>
  </wsgen>
</target>

<!-- Generate the final service WAR file -->
<target name="war-ws-server" depends="create-ws-server-wsdl" >
  <!-- Generate a WAR file! -->
  <war destfile="${ws2rmi.output.war.file}" ... >
    ... (all the previously created files + classpath ...
  </war>
</target>

<!-- Attest to the compilation process -->
<target name="create-attestation" >
    <!-- Chose an AIK -->
    <property name="aik.label"
      value="ddd43906-d0f6-43bb-b906-0f0e1e3b7354" />

    <quote number="${quote.pcrs}"
        ownerKey="..." ownerKeyMode="PLAIN"
        aikSecret="..." aikSecretMode="PLAIN"
        aikStore="${aik.store.dir}"
        destfile="${quote.output.file}"
        pcaCertificate="${pca.cert.file}"
        aikLabel="${aik.label}" />
    <!-- optionally, remove "aikLabel" and include the following
        to go and fetch a new AIK certificate from the PCA
        pcaLocation="http://privacyca:20000/aik" -->
</target>

<!-- Measure and copy the original java interface file  -->
<target name = "measure-input-file" >
  <record name="${log.file}" append="true"  action="start" />
  <extend pcr="${compile.pcr}">
    <fileset file="${interface.class.file}" />
  </extend>
  <record name="${log.file}" append="true"  action="stop" />
  <copy file="${interface.class.file}" todir="${output.dir}" />
```

```
  </target>

  <!-- All put together, this will generate the RMI server, measure
      the interface file and attest.  After this is run, the Web Service
      WAR file must be generated
  -->
  <target name="create-and-attest" >
    <!-- Measure and record the interface file -->
    <antcall target="jar-rmi-server" />
    <antcall target="measure-input-file" />
    <antcall target="create-attestation" />
  </target>
</project>
```

B.2 Prolog Verification Script

The following Prolog script shows the full verification code for interpreting a Prolog system model, as described in Section 6.4 and used in Section 8.2.6.

```
execute_app( A, CS, P, (ID, A, SS, P, OPEN) ) :-
  skolemise(ID),
  programme(A, OPEN),
  findall((A,S), state(A,S,_,_,_), S1),
  findall((C,S), (state(C,S,_,_,_), member(C,CS)), S2),
  append(S1,S2,SS).

possible_next([], []).
possible_next([(ID,A,_,_,OPEN)|APPS], [(ID,A,OPEN)|STATES]):-
  possible_next(APPS,STATES).

iterate_start( A,CS, ML2, RES):-
  execute_app(A,CS,'0',APP),
  append([measure(A,CS)],ML,ML2),
  iterate_all([APP], ML, RES).


iterate_all( APPS, [], APPS).
iterate_all( APPS, [M|ML], RES ) :-
  clean_app_states(APPS,APPS2),
  clean_apps(APPS2,APPS3),
  member(A, APPS3),
  remove_first(A,APPS3,APPS4),
  next_transition(A, M, RES1),
  append(RES1, APPS4, RES2),
  clean_app_states(RES2,RES3),
  clean_apps(RES3,RES4),
  iterate_all(RES4, ML, RES).

clean_apps([],[]).
clean_apps([(_, _, _, _, [])|APPS], APPS2):- clean_apps(APPS,APPS2).
clean_apps([(ID, A, ISS, P, [X|XS])|APPS],
    [(ID, A, ISS, P, [X|XS])|APPS2]):- clean_apps(APPS,APPS2).
```

```
clean_states([],[]).
clean_states([end|XS],YS):- clean_states(XS,YS).
clean_states([X|XS],[X|YS]):- X\= end, clean_states(XS,YS).

clean_open_states([],[]).
clean_open_states([ifopenstates(_)|XS],[]):- clean_open_states(XS,[]).
clean_open_states([X|XS], [X|XS]):- X \= ifopenstates(_).
clean_open_states([X|XS], [X|XS]):- \+ clean_open_states(XS,[]).



clean_app_states([],[]).
clean_app_states([(A,B,C,D,ST)|APPS], [(A,B,C,D,ST2)|APPS2]):-
  clean_states(ST,ST2), clean_apps(APPS,APPS2).

next_transition( (ID, A, ISS, P, OPEN) , M , APPS):-
  clean_open_states(OPEN,OPEN2),
  member(O, OPEN2),
  remove_first(O, OPEN2, OPEN3),
  next_transition2((ID, A, ISS, P, OPEN3), O, M, APPS).

next_transition2((ID, A, ISS, P, OPEN), newstate(X), M,
    [(ID, A, ISS, P, [NTRANS|OPEN])]):-
  from_ids(ISS, SS),
  from_id(X, SS, state(_,_,_,M,NTRANS)).

next_transition2((ID, A, ISS, P, OPEN), parallel([X]), M, APPS):-
  next_transition2((ID, A, ISS, P, OPEN), X, M, APPS).

next_transition2((ID, A, ISS,P, OPEN), parallel([X|XS]), M,
    [(ID,A,NSS,P,[parallel([N|NT])|OPEN])|APPS2]):-
  member(X1,[X|XS]),
  remove_first(X1, [X|XS], XS2),
  next_transition2((ID, A, ISS,P,OPEN), X1, M, APPS),
  member((ID, A, NSS,P,NOPEN), APPS),
  remove_first((ID, A, NSS,P,NOPEN),APPS,APPS2),
  append(NOPEN, XS2, NOPEN2),
  clean_states(NOPEN2,[N|NT]).

next_transition2((ID, A, ISS,P, OPEN), parallel([X|XS]), M,
    [(ID,A,NSS,P,OPEN)|APPS2]):-
  member(X1,[X|XS]),
  remove_first(X1, [X|XS], XS2),
  next_transition2((ID, A, ISS,P, OPEN), X1, M, APPS),
  member((ID, A, NSS,NOPEN), APPS),
  remove_first((ID, A, NSS,P,NOPEN),APPS,APPS2),
  append(NOPEN, XS2, NOPEN2),
  clean_states(NOPEN2,[]).

next_transition2((ID, A, ISS,P, OPEN), choice(XS), M, APPS):-
  member(X,XS),
  next_transition2((ID, A, ISS,P, OPEN), X, M, APPS).

next_transition2( (ID, A, ISS, P, OPEN), app(A2,CS), measure(A2,CS),
    [(ID, A, ISS,P,OPEN) , APP] ):-
  execute_app(A2, CS, ID, APP).
```

```
next_transition2(APP, dothen(end,Y), M, APPS):-
  next_transition2(APP, Y, M, APPS).

next_transition2((ID, A, ISS,P,OPEN), dothen(X,Y), M,
    [(ID,A,NSS,P,[dothen(OPEN3,Y)|OPEN])]):-
  next_transition2((ID, A, ISS,P,[]), X, M, [(ID,A,NSS,P,OPEN2)]),
  clean_states(OPEN2,[OPEN3]).

next_transition2((ID, A, ISS,P,OPEN), dothen(X,Y), M,
    [(ID,A,NSS,P,[dothen(end,Y)|OPEN])]):-
  next_transition2((ID, A, ISS,P,[]), X, M, [(ID,A,NSS,P,OPEN2)]),
  clean_states(OPEN2,[]).

next_transition2((ID, A, ISS,P,OPEN), measure(M), M,
    [(ID, A, ISS,P,OPEN)]).

next_transition2((ID, A, ISS,P,OPEN), loadconfig(CONF),
    CONF, [(ID, A, ISS2, P, OPEN)]):-
  findall((CONF,S), (state(CONF,S,_,_,_)), S2),
  append(S2, ISS, ISS2).


next_transition2((ID, A, ISS, P,[O|OPEN]), ifopenstates(X), M, RES ):-
  next_transition2((ID,A,ISS,P,[O|OPEN]), X, M, RES).

next_transition2(ISS, repeatedly(X), M, NTRANS, APPS):-
  next_transition2(ISS, X, M, [repeatedly(X)|NTRANS], APPS).

remove_dup_m_wrap(X1,Y):-
  reverse(X1,X2), remove_dup_measures(X2,X3), reverse(X3,Y).

remove_dup_measures( [] , []).
remove_dup_measures( [ measure(A,B) | XS ] , [measure(cached(A),B)|YS] ) :-
  member(measure(A,_), XS), remove_dup_measures(XS,YS).
remove_dup_measures( [ measure(A,B) | XS ] , [measure(A,B)|YS] ) :-
  (\+ member(measure(A,_), XS)), remove_dup_measures(XS,YS).
remove_dup_measures( [X|XS] , [X|YS] ) :-
  X \= measure(_,_), remove_dup_measures(XS,YS).



from_id_inner(_, [], []).
from_id_inner(X, [state(A,X,B,C,D)|XS], [state(A,X,B,C,D)|RS]):-
  from_id_inner(X,XS,RS).
from_id_inner(X, [state(_,Y,_,_,_)|XS], RS):-
  Y \= X, from_id_inner(X,XS,RS).

from_id(X,Y,Z):-
  from_id_inner(X,Y,RS),
  get_lowest2(RS,Z).

from_id((A,X), state(A,X,Y,Q,P)):- state(A,X,Y,Q,P).

get_lowest2( [state(A,X,Y,Q,P)], state(A,X,Y,Q,P) ):-
  state(A,X,Y,Q,P).
```

```
get_lowest2( [state(_,_,N,_,_)|SS], (state(_,_,N,_,_)) ):-
  get_lowest2(SS, (state(_,_,N2,_,_))), N2 > N, !.
get_lowest2( [_|SS], R ):- get_lowest2(SS, R).

from_ids( [] , []).
from_ids( [X|XS] , [Y|YS]) :- from_id(X, Y), from_ids(XS,YS).

remove_item(_,[],[]).
remove_item(X, [X|XS], Y) :- remove_item(X,XS,Y).
remove_item(X, [A|XS], [A|Y]) :- A \= X, remove_item(X, XS, Y).

remove_first(_,[],[]).
remove_first(X, [X|XS], XS).
remove_first(X, [A|XS], [A|Y]):- A \= X, remove_first(X, XS, Y).

skolemise(T) :- var(T), gensym(t,T), !.
skolemise(T).
```

Appendix C

Trusted Computing and Provenance: Better Together

This appendix includes a copy of the paper 'Trusted Computing and Provenance: Better Together' [124] with some background sections omitted. This was co-authored by Dr Andrew Martin. My contributions to the paper include the main content of the literature review, remote attestation-based provenance system, missing components and the comparison between research fields.

The paper is included to demonstrate that trusted computing and attestation would be useful in a real situation requiring trustworthy services. Furthermore, as discussed in Section C.5 trusted computing and provenance research overlaps considerably. Both fields discuss the collection of integrity information and have the same issues regarding the management and collection of frequently-changing reference data. This appendix shows the related research and how an attestation-based infrastructure could provide a trustworthy way to implement secure provenance.

Abstract

It is widely realised that provenance systems can benefit from greater awareness of security principles and the use of security technology. In this paper, we argue that Trusted Computing, a hardware-based method for establishing platform integrity, is not only useful, but immediately applicable. We demonstrate how existing Trusted Computing mechanisms can be used for provenance, and identify the remarkable similarity and overlap between the two research areas. This is accomplished through presenting architectural ideas for a trusted provenance system, and by comparing the respective requirements and capabilities of trusted systems and provenance systems.

C.1 Introduction

Provenance information is essential for maintaining the integrity of scientific results, particularly those that are difficult to reconstruct independently. Through it we can verify the origins of primary source data, how it has subsequently been processed, and create a complete set of instructions as to how to recreate the final results. Many systems exist to support the collection of provenance information in e-Science [19], providing some of the following functionality [193]:

- Find 'the sources of faulty, anomalous processing outputs.' [19]
- Allow judgement of data quality [139]
- Support the replication of results
- Maintain the correct attribution of data
- Augment results with additional experimental context
- Enhance trust in scientific results [73]

We postulate that as such provenance information becomes more widely available, and more reliance is placed upon it, there will be an increasing need for strong guarantees of its accuracy: in information security terms, many will be concerned with provenance *integrity* and therefore with tamper-proofing the systems which record and process such information.

Meanwhile, research in Trusted Computing is attempting to provide trustworthy platforms, where the integrity of data storage and program execution can be assessed (and enforced) remotely. The aim is to provide security and assurance despite the presence of malicious software. Longstanding research and development efforts mean that implementations using a mix of hardware and software mechanisms are available today, with the hardware components quickly becoming ubiquitous in commonplace computing platforms. In this paper we argue that these new security technologies provide many of the features that provenance systems require. And because they have security as a primary design goal, can be used to implement *trustworthy* provenance systems with little additional effort or modification.

The rest of this paper is structured as follows. In Section C.2 we provide a brief overview of provenance. We then discuss the need for trusted provenance in Section C.3. Following this, we outline a provenance architecture built on Trusted Computing technology and standards, highlighting how existing software and specifications can be used, what modifications would be necessary, and the advantages of doing so. In Section C.5 we show that several research problems (and proposed solutions) are common to both Trusted Computing and provenance, and that both areas can benefit from collaboration with each other. We then mention some of the remaining challenges in implementing trustworthy provenance, and finally Section C.7 presents our conclusions.

C.2 Background

C.2.1 Provenance

'Provenance' or 'lineage' generally refers to information that 'helps determine the derivation history of a data product, starting from its original sources' [193]. In other words, a record of where data came from and how it has been processed. This is particularly applicable to e-Science, as the quality of experimental data is important. Indeed, Moreau et al. [139] state that:

> 'In an ideal world, e-science end users would be able to reproduce their results by replaying previous computations, understand why two seemingly identical runs with the same inputs produce different results, and determine which data sets, algorithms, or services were involved in their derivation.'

The assertions made in a provenance system (*p-assertions*) can be categorised in many ways. Cheney et al [39] refer to 'how,' 'why' and 'where' statements, and Vázquez-Salceda et al. [229] define 'interaction,' 'relationship' and 'actor state' categories. The latter being information about the state of a participant in a workflow or process that manipulates or creates the original data.

There are several existing systems defined for recording process provenance. We skip a full review, as details can be found in survey papers [193, 19]. The First Provenance Challenge [140] is a good starting point for comparison.

Security and data integrity are becoming increasingly relevant to provenance, as researchers become more aware of the threats posed. Several proposed systems use kernel and file system-level monitoring to protect the collection of provenance information [228, 185], removing it from the user's control. Hasan et al. [87] provide a thorough analysis of threats to provenance systems, and have proposed a system using encryption and chained signatures to provide integrity protection. The authors make a good point that without a 'trusted pervasive hardware infrastructure,' there will always be potential attacks. We will demonstrate in this paper that such an infrastructure can readily be provided by Trusted Computing, and therefore believe that our paper is complementary to their work. Similarly, Zhang et al. [243] use hash chains to provide *tamper-evident provenance* in databases, and tackle the issue of providing audit logs of compound objects rather than just for a linear sequence of operations. They state that the use of trusted hardware is 'impractical' due to the loosely-organised nature of provenance collection and sharing. We believe that Trusted Computing is cheap and pervasive enough to avoid these issues in many scenarios. Tan et al [203] have also listed several security requirements for provenance, discussing signatures on p-assertions in order to provide integrity guarantees, as well as accountability. Braun et al. [23] discuss the challenges of securing provenance data when it may contain sensitive or confidential information. We are more concerned with the *integrity* of provenance information in this paper.

C.3 The Case for *Trusted* Provenance

Perhaps one of the most significant reasons for keeping provenance information is to provide assurance in the quality of scientific results [73]. This usually means protecting against unintentional error, or malfunctioning equipment. However, for high-profile science, such as climate change and pharmaceuticals, the risk of intentional, malicious intervention becomes just as important. In these situations there are threats from outside – organisations and individuals wishing to manufacture results supporting their interests, or discredit research that damages their products. Separately, and perhaps more invidiously, the user/researcher may have their own motivation for falsifying data. We believe that provenance systems should be able to identify and record these threats. But to do so reliably, records must be robust and secure. They must be highly tamper-resistant, making any successful attack on their integrity infeasible. If provenance records are not protected, then they cannot provide convincing evidence of the quality of the data itself.

Clearly, the provenance of results relies upon both hardware and software—as those who encountered the Intel Pentium floating point bug [95] learnt to their cost. In a massively distributed system, such as that provided by a grid or cloud computing scenario, such concerns are all the more important—but potentially also give rise to a significant overhead in metadata management. Such systems may be located outside the user's own department, perhaps a different university or even on another continent. They are subject to the oversight of many unseen administrators, hardware changes, and software upgrades and patches.

Moreover, the scope for malicious interaction is great: too many individuals are involved, and so reliance upon informal trust relationships is infeasible. Even where all those participating are honest, there remains the possibility of viruses and trojans.

Such concerns are illustrated well by the challenges of 'public resource computing' projects such as climate*prediction*.net [199]. By distributing computational tasks to hundreds of thousands of users around the world, substantial resources can be brought to bear upon a task like climate modelling—but the results are open to fabrication, or the introduction of systematic bias. The duplication of tasks can help to reduce this risk, but at the cost of effectively reducing also the amount of computational power available. Although one may *hope* that well-managed grid resources will give more reliable results, as the value and impact of those results rises, the need for supporting evidence grows also.

The situation therefore seems quite hopeless: we are in a computing scenario in which we must place a high degree of trust in every possible processing platform, without any meaningful guarantee of trustworthiness. This leaves us open to manipulation, and we cannot consider reported provenance information any more reliable than the reported data. This is made worse by the mutable nature of software and data – it is too easy for a malicious party to alter programmes and records to create believable forgeries. We require some way to retake control, without losing the advantages of distributed processing. The simple addition of extra layers of software controls does not necessarily solve the problem, nor even raise the bar significantly, if the attacker has sufficient motivation.

Here is a role for Trusted Computing and secure hardware. Designed to provide a small,

internal 'trusted third party,' the Trusted Platform Module can be used to record and report the state of the computer in which it is embedded. This is designed to be immune to attack by software (which should eliminate the threat of malware) and can provide exactly the evidence we require that a computer has not been tampered with. If used for provenance, it means that any result processed with illegitimately modified software would always be recorded as such. The means for enhancing platform trust and for collecting provenance data are closely aligned, and can therefore be provided by the same mechanism. The only way to create false records would be to tamper with the hardware itself, an extremely expensive and time-consuming task, beyond the capabilities of most. Having identified that distributed scientific experiments face significant threats, and are performed in low-assurance situations, it seems essential that provenance data be further protected to retain the quality and trustworthiness of computational results.

C.4 Remote Attestation as a Provenance System

Integrity measurements seem immediately applicable to two requirements of provenance: identifying which results have been affected by a known software or hardware error, and for accurately reproducing results. Without knowing the exact versions of software that were used, neither of these things will always be possible. In the language of some provenance research [39], it can help answer questions about 'how' data has been modified and unambiguously identify 'where' it originated. Information about the execution state of a processing node can be considered 'actor state' information in the categories discussed by Vázquez-Salceda et al. [229].

Remote attestation and provenance systems appear to use similar techniques to solve related problems. Because of the similarities between the two fields, in this section we present a provenance architecture based entirely on available Trusted Computing software and hardware. While this is not a complete system, and does not provide answers to many provenance questions, we demonstrate that certain aspects of provenance can easily be implemented in this way, with the built-in benefit of high assurance.

C.4.1 An attestation-based provenance architecture

We assume a service-oriented infrastructure, perhaps implemented as a grid or cloud, with a number of remote platforms performing computations (see Figure C.1). Each machine has a Trusted Platform Module and, when initially added to the network, they are issued an Attestation Identity Key (AIK), signed by a certificate authority (Privacy CA). This key will be used for subsequent attestations, and uniquely identifies the platform. At this time, an administrator will record the machine's original hardware details and software measurements. Much of this process is defined in the Trusted Computing specifications [207]. The platform itself uses software that supports authenticated boot, and the TPM will therefore record all running executables, usually in the first twelve platform configuration registers. This, along

with the job request and result, will be the provenance data captured by each platform.

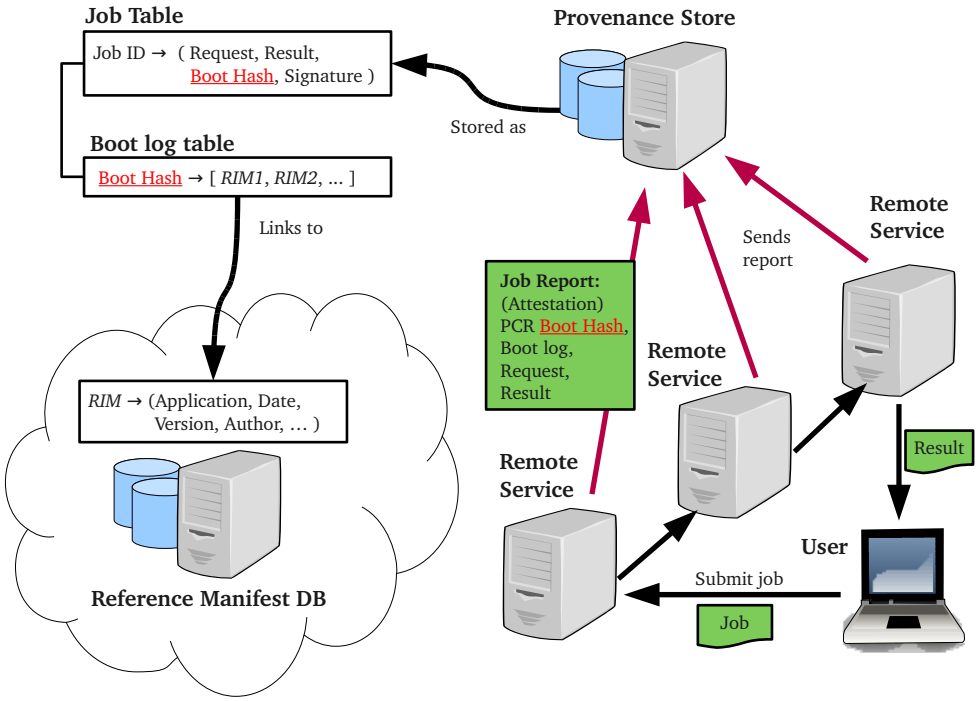

Figure C.1: Diagram of an attestation-based provenance architecture. Remote services process results and attest to the provenance store, which saves and links the measurement logs to a TCG-defined Reference Manifest Database.

When the platform receives a job, it does the following:

1. Measure a hash of the received job (or, if it is a web service, the incoming request) into PCR 11 of the service's TPM.
2. Execute the job
3. Hash and measure the job result (or reply message) into PCR 11.
4. Sign PCRs 0-11 with the Attestation Identity Key and send them to the provenance store, along with the measurement log. This log contains a list of all the values extended into each PCR.

The provenance store will receive regular reports from the processing platforms, consisting of attestations, measurement logs and the results of processed jobs. This information will be connected to other sources of provenance data, such as the workflow description. The report signatures will be verified, and the reported PCR values will be checked to make sure that they correspond to the log. If either of these steps fail, this implies a software error (or malicious intervention) and jobs should be rerun on a different machine. In either case, a copy of the attestation and log should be recorded in the provenance store.

The contents of the measurement log for PCRs 1-10 will contain a list of every piece of software executed on the platform. This information needs to be stored for every job. However, if exactly the same software has been run as on a previous attestation, then the final PCR hash will be identical. In this case it will suffice to list all the software once, and link to it from the provenance database. This means that the majority of entries into the provenance store just consist of the attestation itself, and will therefore be extremely small - only a few 20 byte hash values. Of course, a full list of software and hashes (RIMs) will need to be maintained somewhere. In the TCG model, this would be a Reference Manifest Database, and there are well-defined schemas for each entry, as well as protocols for keeping the database up to date and accessible [207]. Such a system is given in Figure C.1. We note that compacting provenance information through hash chains has been discussed before by Hasan et al. [87] and Zhang et al. [243].

Every time a new patch is loaded onto a service machine, this will result in a new hash and therefore a new chain of trust being stored. However, the storage overhead should remain small. Based on data from 2006 to 2009, a typical web service can expect to be updated only around 22 times a year, with under 500 new hash values [122]. This is trivial amounts of data, particularly as most machines will be running near-identical sets of software.

C.4.2 Software and hardware details

Almost all the software and hardware required for implementing the described system is freely available today. TPMs are installed in many business notebooks and servers, and the Linux kernel now supports Authenticated Boot [180]. TCG compatible software stacks exist, including the JTSS [101] in Java and TrouSerS [219] in C++. These make writing programs that use the TPM straight-forward. Reading and extending PCR value in Java, for example, requires the JTSS libraries, around ten lines of initialisation code, and then just one line to actually read or extend. Privacy CA software and an integrity-measuring Java Runtime Environment is also available on the JTSS website. The OpenPTS [155] 'Platform Trust Service' project provides the infrastructure for creating integrity measurements, collecting RIMs and connecting to an external software repository. Furthermore, web service protocols are already defined to maintain compatibility with WS- standard [142]. In fact, the only custom changes to software would involve updating the service interface (or grid middleware) to measure incoming requests and outgoing results, and to send attestations to the provenance database. This should be straight-forward.

C.4.3 Advantages

This infrastructure immediately provides several advantages to ad-hoc reporting of platform information. Forging integrity reports is infeasible, thanks to the secure key storage provided by the TPM. Because AIKs are stored in the TPM, and cannot be disclosed, it would be extremely difficult to assert that a different machine produced the result. Because of the platform configuration registers, authenticated boot process and software support, it also

should not be possible to claim that a different version of a particular piece of software was being used. And as we are measuring the input and output, we can be sure of exactly which software was used to process a particular job, and what output it produced. This means that the attested information meets the requirement given by Groth et al. [82] for high integrity p-assertions. Furthermore, attestations can potentially be created autonomously, at any point in time, a requirement defined by Groth and Moreau [81]. Thanks to the software and hardware developed, attestations must also produce a complete record of all software used, at every stage of platform boot and throughout its use. This includes firmware, drivers and shared libraries, potentially a more comprehensive (and accurate) list than other systems are capable of producing. Jobs can be time stamped, through use of the TPM's tick counter. In addition, the presence of TPM hardware is an opportunity to improve the security of credentials, as keys can remain protected from the rest of the system.

All of these benefits come merely by leveraging existing Trusted Computing techniques. This is significant because it uses a single technological base to give advantages for both short-term trust and long-term provenance.

C.4.4 Missing components

The system discussed can be enhanced considerably. There are some obvious missing features and functionality. The attested information gives only the *execution state* of the platform, at the level of applications run by the operating system. Nothing more fine-grained can be reported. Current implementations will also not re-measure a program, which means that the precise ordering of execution will not be preserved. Other missing information includes configuration files, environment variables, generated code, and load information (free disk space, processor utilisation, and so on). However, it would be relatively straight-forward to include all of this information, as it should only be necessary to modify middleware or adapt an existing system such as PASS [143] or Provenance Aware Condor [171]. 'Semantic Attestation' also aims to solve this problem [85].

Perhaps more importantly, integrity reports would need to be mapped to the rest of the information produced by a full provenance architecture. This includes records of *who* accessed data, what the overriding purpose of the request was, and how each individual platform was used in combination for a full process workflow. Such information must (in part) be provided by the end-user, and will need to reference the generated integrity measurements.

The above system does not provide any easy mechanism for recreating results. While it would be possible to guarantee that two results came from precisely the same software execution, if hardware or software is changed subsequently, there is no way to re-run with the original versions. We suggest that this could be implemented through saving and restoring virtual machines. Such an approach would be compatible with the Eucalyptus cloud computing system [152].

It would also be necessary to include custom developed software in the Reference Manifest Database (RMDB), and extend the TCG schema [220] to include information about how it was built. The provenance store would also need to be modified suitably to work with a RMDB

and to support searches for all results that used certain pieces of software.

C.5 Provenance and Trusted Computing Research: Producing the Same Solutions

Despite the lack of interaction between Trusted Computing and provenance communities, there is a great deal of overlap in current research. The security industry is looking for better methods for monitoring a platform's behaviour, a task that provenance systems already focus on: in many security contexts, *prevention* is infeasible or prohibitively expensive; *detection* is often a viable alternative. Detection of anomalies is therefore of great interest. Moreover, provenance research is looking to increase the trustworthiness and integrity of records [86, 203], a well-established problem in security. In this section we identify common areas of work, and look at the related (but perhaps unknown) literature.

C.5.1 Related research

Both provenance and Trusted Computing are concerned with monitoring and reporting the state of a machine used for some high-value (or high risk) function. Huh and Martin [93] look to provide more detail by intercepting and securely logging I/O requests. In the provenance domain, PASS [143] provides logging through hooking system calls, and Clifford et al. [45] provide runtime execution logging. Reilly and Naughton [171] have similar ideas, but use an extension to Condor to perform transparent logging. The work by Huh and Martin will provide a higher assurance, but the approach taken by Reilly and Naughton may result in more useful data. A common theme in this section is that Trusted Computing research currently focuses on creating comprehensive and high-integrity results, whereas provenance systems are better at extracting exactly the information considered relevant, and are not constrained by security issues.

Tracking data usage is an important functionality of a provenance system, but has also been approached many times in Trusted Computing research, with digital rights management in mind. Proposals by Nauman et al. [147] would enable 'measurement, storage and reporting of the attribute update behavior' for a data item at a remote platform. This is part of the functionality required for provenance [139], and we can imagine it being integrated with data derivation graphs. Provenance, access control and usage control have been linked before, notably by Ni et al. [150].

The construction of custom executables and their history has been approached by both areas. In earlier work, we have used integrity measurement to measure the compilation process in order to produce a trustworthy compilation certificate for an arbitrary programme [121]. This is similar to the 'Transparent Make' functionality provided by Vahdat and Anderson's TREC lineage system [228] and PASS by Muniswamy-Reddy et al. [143]. Again the goals are different, but both allow users to identify how an executable was formed and what its dependencies are. The similarities are notable, as TREC allows dependencies to be specified through Make files,

and we provided the same through ANT build scripts. However, TREC is general-purpose and has many other applications, operating constantly through a kernel module intercepting system calls. This makes it more suitable for constant use. Our compilation certificates, on the other hand, are verifiable and considerably more trustworthy, but must be run independently of normal processing. Overall, there is a clear requirement for greater information about compiled software in both fields.

For full provenance information, we need to know how data is stored. The Trusted Compting Group have standards for 'Trusted Storage' [222], providing features such as disk-encryption, authentication and logging. The logging use case specified by the TCG has forensics and auditing in mind [216]. Again, this is similar to the requirements for provenance [86] - we would like the ability to go back through records and establish whether tampering or unauthorised access occurred. While we are not aware of any hardware-based related provenance research, secure and audited storage through file system and kernel support has been mentioned frequently [228, 143, 185].

Both Trusted Computing and provenance systems tend to involve modification of existing middleware. Condor has been modified by researchers in both fields [171, 146] looking to add security and lineage through monitoring and assessing individual platforms. Löhr et al. [116] proposed new modifications to grid middleware for enhanced trustworthiness, and Frew and Slaughter [67] have demonstrated provenance in the ES3 system. Similarly, Trusted Cloud Computing [184] and Provenance-Aware Cloud Computing [144] have both been discussed recently, as well as Service Oriented Architectures [15, 203]. This implies that not only are similar problems being solved, but in the same context and using the same underlying software and systems.

C.5.2 What provenance can gain from Trusted Computing

We have already discussed the motivation for trusted provenance, but the use of Trusted Computing has many potential advantages. Going beyond the enhanced security and assurance, Trusted Computing research typically considers a much wider range of factors that can affect system behaviour. This includes CPU architecture, use of virtualisation, protocols and software. By taking advantage of this thoroughness, provenance systems can be more comprehensive and may identify hidden factors [67] that will later be useful. Furthermore, as security problems receive greater attention and funding, it seems sensible to take advantage of the new hardware and processes that are being implemented and re-use them for provenance. Trusted Computing is led by a group of companies (such as IBM, Intel, AMD, and Microsoft [214]) with significant resources. By introducing provenance as a requirement, we believe that many of the tools being developed and used for security can become immediately useful for provenance.

C.5.3 What Trusted Computing can gain from provenance

Provenance research can be used by Trusted Computing researchers to enhance system security and auditing. Integrity reporting systems lack a framework for evaluation, and require a way

of interpreting results. The provenance community is more aware of systems involved with semantic representation and metadata, which could be the solution to this problem. Both fields also have the problem of reporting too much data, and deciding how best to filter it. We suspect that the provenance community are further ahead in storing and querying this kind of information.

Trusted Computing suffers from another problem that is better understood in provenance: how to deal with incomplete data. Remote attestation can only give details of the current state of a platform, not historical data. Regular attestations, such as those mentioned in our proposals in Section C.4 can provide a better history, but what should happen when a record is missing? How should this scenario be recorded? Provenance systems are already designed to work with incomplete information and composite data sources.

Finally, provenance seems like an excellent use of Trusted Computing, particularly as many of the criticisms of Trusted Computing are less relevant. The integrity reporting approach has been criticised as being fragile when used to make access control decisions, as any missing software in the reference database will result in the platform being denied access. However, in provenance, runtime decisions are not as important as storing a history for later use, so this fragility is less important.

C.6 Challenges

With all the similarities we have listed, there are still some challenges. The research directions are different: the Trusted Computing researchers are currently focused on improving security through advances in cryptographic algorithms, and isolation mechanisms. In comparison, semantic consistency and querying are perhaps more important for provenance. Furthermore, in provenance the goal is to gain extra information for later analysis, and to improve scientific results. And while security can be considered an enabler of new functionality, many still believe it to be just about *preventing* bad things from happening.

Scientific results will face different threats and attacks than many other distributed computing techniques, and a significant challenge is making sure that security does not reduce usability. In many cases, there may be a relatively low risk to the data, and this should be reflected in the security architecture. As a result, the use of Trusted Computing should be as transparent as possible, and require as little effort for users and developers of applications (often the same people). An open challenge is developing a systematic methodology for creating applications that support provenance and provide high assurance. This may involve combination of recent work on PrIMe [137] and security development lifecycles [114].

Performance is another issue that could prevent the adoption of both provenance and security technology. The TPM is a low-speed chip, and cryptographic operations (such as attestation) are relatively slow. The authenticated boot process also impacts on performance [180]. However, new versions of the TPM may be faster, using symmetric cryptography [221] and it is likely that the hardware manufacturers will be able to increase performance in the future. For the time being, there has been research looking at improving efficiency [33], and the

provenance architecture we outlined would require only one attestation per submitted job and platform.

Trusted Computing also relies upon a public key infrastructure, for certifying attestation keys and identifying platforms. We have not explored the trust management issues in depth in this paper, and there will undoubtedly be issues in maintenance and implementation. Fortunately, scientific grid computing is one domain with experience in key management on a large scale, and we are optimistic in solving such problems.

C.7 Conclusion

Many of the concerns addressed by Trusted Computing relate to immediate and short-term policy enforcement ('shall I share this secret with that software, on that platform?'). Many provenance issues are of a more long-term nature ('where did this come from; how was it processed?'). Yet these are highly-related topics because they both rely upon the unambiguous (and tamper-proof) identification of hardware and software.

Existing Trusted Computing systems *already* provide much of the required functionality, and in a way that provides high assurance and makes forgery infeasible. Furthermore, with the introduction of security systems using Trusted Computing, the hardware becoming available, and software libraries and OS infrastructure, the basic capabilities for collecting highly-assured provenance data are being built. We have identified several places in which the two research areas overlap, such as logging, monitoring, compilation history and secure storage.

We have argued that there is a natural synergy between the two areas of research, an overlap in both the goals and the technologies for achieving them, and a strong prospect for combining the two to give rise to *trusted provenance*.