Trustable Remote Verification Of Web Services

John Lyle

Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford, OX1 3QD
john.lyle@comlab.ox.ac.uk

Abstract. Service Oriented Architectures currently provide little or no evidence that each remote component has been implemented correctly. This is a problem for businesses hoping to exploit the potential benefits of SOA. We present a technique called Trustable Remote Verification, which lets providers create behavioural guarantees of their web services. Our approach is flexible, using Extended Static Checking for verification and has the significant advantage of requiring no additional trusted third party.

1 Introduction

As applications and services move online, we are implicitly expected to place more trust in the developers and providers of these systems. But how do we know if a remote service is trustworthy? Bugs and vulnerabilities will continue to exist, and while dedicated providers might be considered more capable of administering systems than end users, they also become valuable targets for attack. As a single point of failure, a poorly implemented service will affect everyone who relies upon it. We therefore need some way of establishing trust in remote systems.

There are several issues to overcome before being able to trust an online application. Fundamentally, the users of a web service have no information about the quality of its implementation. Furthermore, most services only have a WSDL[1] description, which contains little semantic information, only method names and basic data types. Without a more detailed behavioural interface, trustworthiness is difficult to establish. If we do not know how it *should* work, whether or not it has been programmed correctly becomes irrelevant. Finally, software installed at a service will frequently be patched and upgraded, usually without warning or notification. As a result, a relying party will potentially need to reassess it before every request.

We have developed a solution to some of these problems which uses trusted computing and tools for specifying and verifying Java programs. A brief overview of these in given in Section 2. In Sections 3, 4 and 5 we refine the problem, introduce the idea of Trustable Remote Verification, and present details of our prototype. It is then evaluated and compared against existing solutions in Section 6. Future work is discussed in Section 7 and in Section 8 we conclude.

2 Background, Definitions and Related Work

2.1 Remote Attestation

Remote attestation is an authentication technique for trusted computing platforms. It uses the 'TPM Quote' operation to create a signed record of the attesting computer's Platform Configuration Registers (PCRs)[2]. These are intended to be used to record the platform's boot process, including the bios, bootloader, OS and applications. A quote can therefore be considered trustworthy evidence of what software has been run on the attesting platform. This is a valuable piece of information, as it potentially gives a remote user the ability to identify and make a judgement about the software that is running.

In a basic server implementation, PCRs are held on a separate chip, the Trusted Platform Module (TPM), and store a 20 byte SHA-1 hash value. These values can only be modified by software through the `extend(..)` command. This appends the current PCR value to the supplied input, hashes it, and stores the result in the PCR. A PCR value therefore reflects a chain of individual hashes. The TPM Quote operation takes a number of these PCR values and signs them with the private half of an *Attestation Identity Key* (AIK), which is held within the platform's TPM. This AIK must have a credential, issued by a trusted third party (a 'Privacy CA'), which typically states that the platform has a valid TPM[3]. Upon receiving a quote, therefore, it is necessary to check that the AIK used has been certified by a trustworthy Privacy CA. Quotes can also contain a nonce, set by the challenger, in order to guarantee freshness.

On platforms which support *authenticated boot*, every piece of code executed in the boot process is hashed and extended into a PCR. This is done sequentially (a *chain of trust*), with each preceding step extending a measurement of the next before allowing it to execute. This means that a malicious piece of code cannot be run without its identity first being recorded. So long as every step in the process is trusted to perform measurements faithfully, it is possible to attest that a platform is only running known pieces of software. Remote attestation is therefore about reporting system *integrity measurements*, as the modification of any executable will be noticed. This is obviously attractive from a system security perspective, as it becomes possible to identify a machine which has been infected with a virus or rootkit.

However, attestation does not necessarily give any indication of a platform's *security state* but rather its *execution state*[4]. Sadeghi and Stüble[5] introduce the notion of 'Property-Based Attestation' (PBA) in order to simplify the process. In PBA, platforms provide a list of guaranteed security properties, rather than just a binary integrity measurement. This can be implemented in a number of ways, but generally relies upon at least one party being able to match the PCR values to security properties, and then issuing certificates to this end. PBA is a level of indirection which can take some of the burden from the attestation requester.

2.2 Web Services

Web services are a well-established approach to creating large, component-based applications[6]. Services communicate and describe themselves using standard data formats, such as SOAP[7] and WSDL, and have interoperability as a primary requirement. Each service offers some kind of useful functionality, but 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 be dynamically avoided by choosing alternative services where necessary.

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. 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.

2.3 JML and Design by Contract

The Design by Contract (DbC) approach advocates having a 'precise definition of every module's claim and responsibilities'[8] in order to create reliable and, importantly, *reusable* components. This is exactly what we would like to achieve with web services. 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'[8]. Several annotation languages exist for the DbC methodology, including Eiffel, Spec# and JML, the Java Modeling Language. JML[9] offers other language features, including specification of exceptions, non-null annotations and class ownership. A simple example of JML can be found in Figure 1.

2.4 ESC/Java2

The annotations introduced by Design by Contract can be used for automatic source-code checking. This is usually performed with a static analyser, which attempts to verify assertions without ever executing the code. One such analyser is ESC/Java2[10], an extended static checker for Java and JML. It translates code and annotations into logical terms, and then runs these through the Simplify theorem prover, producing either a counter example or an 'ok' result.

3 A Basic Web Service Behavioural Attestation Model

For the purposes of this work, we assume a simple scenario where a service requester (R) wants to use a web service (W). Before doing so, R wants to find

```
/*@ 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) {
      ...
}
```

Fig. 1. An example web method, complete with JML annotations. Two outcomes are specified: 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.

out two things: what software is being run at W and what that software promises to do. The first part can (in theory) be achieved through remote attestation, but the second is more difficult. The approach we have taken is to add JML annotations to the service end point. This means that each web method can include pre- and post- conditions, as well as other formal properties. Although only Java services are supported, similar methods could be used with Spec# for .Net services, or any other Design by Contract language.

However, we have yet to show that the annotations attached to a service are really implemented. In this basic architecture, a *Trusted Compilation Service (C)* fulfils this role. The source code (W_{src}) of W is compiled and checked against its annotations (W_{ann}) by C, using a method such as model checking. A certificate is then produced by C which contains a hash of the compiled application (W_{bin}), a list of any dependent libraries used in the compilation process and an assertion stating which program properties hold. W can then present this certificate to R, along with a fresh remote attestation[1], and R will have a much greater level of confidence that W has the promised behaviour.

There are a number of assumptions. R must trust that C will do a thorough analysis of the source code at W. W must trust C with all the source code, which may not be possible. More fundamental assumptions include:

[1] The fresh attestation must, of course, contain an entry which is identical to W_{bin}. If it does not, then the certificate produced by C cannot be used to validate this service.

– C must be capable of analysing the source code at W. This is not a trivial task, and may depend on the availability of numerous code libraries, operating system features, and so on.
– The middleware and OS running at W must also be trusted by C and R.
– All important configuration settings must be either made available to C or R. This might be undesirable if they contain passwords or confidential settings.
– The middleware and OS must be *attestable*. Each part of the software stack must support integrity measurement. It must be possible for R to receive a remote attestation from W and interpret it. This implies the existence of an integrity management infrastructure, which has a list of 'known-good' pieces of software. Every binary running at W will need to be on this list.

Overall, this is a proposal for using Remote Attestation to establish meaningful properties of a web service. However, there are a number of drawbacks, including the requirement for a trusted third party, C. The rest of this paper introduces Trustable Remote Verification, which allows us to remove C altogether.

4 Trustable Remote Verification

There are generally two approaches to program verification. It can be done by a trusted third party, but they may charge a high price for their services. They are also a target for attack, and should be avoided if at all possible. The alternative is to verify an application locally. If source code can be inspected before compilation, any errors can potentially be spotted before it 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[11], 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 be considered appropriate and instead we require some way to let the *provider* perform verification, and then prove to users that they have done so.

4.1 Overview

Trustable Remote Verification (TRV) uses TPM attestations as credentials. The service provider performs program analysis using a local machine (the verification platform, V) and then attests the result. To do this, V must be booted into a trustworthy OS, which measures every step of the process and extends each into a PCR. After authenticated 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 1 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

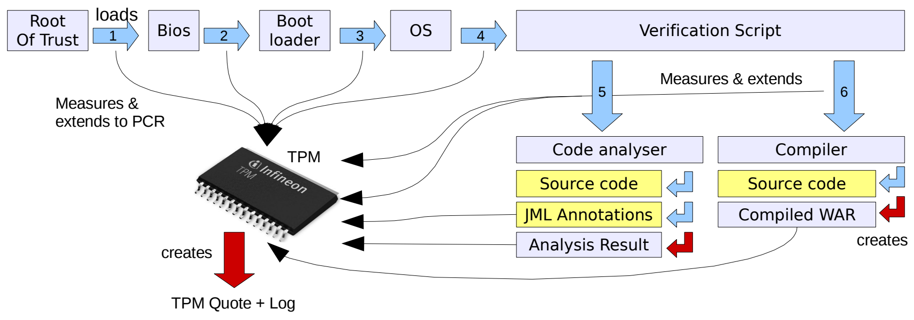

Fig. 2. An overview of the Trustable Remote Verification process, showing the order of execution and all items measured into PCRs.

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 verification process, a quote (V_{quote}) is produced which contains two sets of PCRs, holding measurements:

$$V_{quote} = \left\{ \begin{array}{ll} PCR\ 0..15 = \{\ [\text{ boot process at } V]\ \} \\ PCR\ 21 \quad\ = \{\ TV, TC, W_{ann}, TV_{res}, W_{bin}\ \} \end{array} \right\}_{AIK_V} \tag{1}$$

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.

At runtime, the service provider attests in the normal way, creating another quote:

$$W_{quote} = \left\{ \begin{array}{ll} PCR\ 0..15 = \{\ [\text{ boot process at } W]\ \} \\ PCR\ 21 \quad\ = \{\ L_{bin}\ \} \end{array} \right\}_{AIK_W} \tag{2}$$

Where L_{bin} is the measurement of the binary that has been loaded at service runtime and will be accepting requests. In order for the V_{quote} credential to be useful, L_{bin} must be equal to W_{bin}.

4.2 Assumptions

Trustable Remote Verification relies on several assumptions:

- The platform performing verification has a valid TPM which has not been tampered with.
- A PKI infrastructure for issuing AIKs exists and all platforms have valid AIKs and certificates available to them.

– 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 run without any user interaction, and be trusted to work properly by the client. In our proof-of-concept implementation we have JML annotations as a contract and use ESC/Java2 for verification.
– There is a simple operating system and boot loader, again trusted by the client, which the verifier can run on without interference. This will measure all programs executed on the platform, and is itself measured as part of the boot process.
– 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.

4.3 The New Chain of Trust

TRV decouples the process of certification and application execution. This means that once the web service binary has been verified, it can be run on any service which supports authenticated 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).

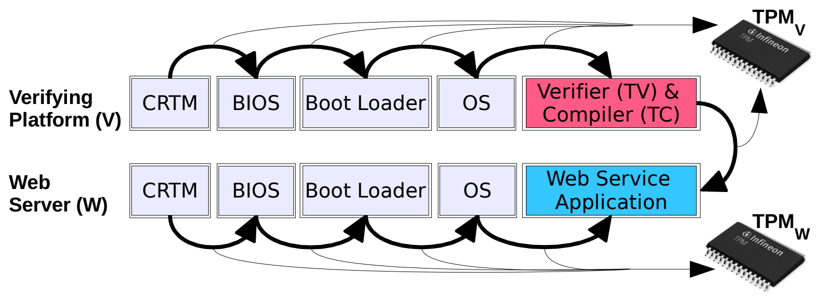

Fig. 3. The chain of trust for Trustable Remote Verification, showing execution order and measurement storage.

5 Prototype Implementation

We developed two parts of this system: the credential-creation stage on V and requester validation stage at R. In our prototype, 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[2] file which is run from the Glassfish Application Server. We did not implement authenticated boot for either the verification or web service, but several potential candidates exist for this purpose[12].

5.1 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 authenticated 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 archived created by compiling the service.
   – The output of running ESC/Java2.
5. A quote (V_{quote}) is created, signed by the AIK, containing 2 sets of PCR values. One has the trustworthy OS and application measurements and the other has all the measurements made by the verifier.

Additionally, an archive 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. This 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.

$$V_{arch} = \{\ V_{quote}, [\text{ libraries }], W_{ann}, TV_{res},\ \text{log}\ , [\text{ config files }]\ \} \tag{3}$$

The compilation stage and WAR file creation is complicated by incompatibilities. Java web service annotations are only valid in Java 1.5 and above but ESC/Java2 can only interpret version 1.4 source code. As a result, the service must be written in Java 1.4 and then wrapped by an automatically-generated Java 1.5 front-end class.

[2] Web application archive `http://java.sun.com/developer/technicalArticles/Servlets/servletapi/`

5.2 Credential Validation Steps

The service requester, R, must obtain and verify the credential that was created using the steps in Section 5.1. This requires the requester to download V_{arch} and obtain a fresh attestation of the web service's current configuration, W_{quote}, and then do the following:

1. Check that the software running on W, as reported in W_{quote} is trustworthy.
2. Check that the currently-running web service application, L_{bin}, matches the verified service identity W_{bin}, included in V_{arch}.
3. Check the AIK used to sign W_{quote} and V_{quote}.
4. Check the freshness of W_{quote}. V_{quote} is not time dependent, so a freshness check is unnecessary.
5. Check that V's OS and boot process are trustworthy.
6. Check that the verification program (including TV and TC) is trustworthy. We imagine checking against a public list of verifiers (with available source code) which are known to be sound and complete.
7. V_{arch} contains a log of the verification process, which will need to be checked against the PCR values held in V_{quote}.
8. 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. With our sample implementation, some of the checks described are performed manually, but it would be feasible to create automated tools. Part 1 and 5 require an integrity management infrastructure, such as IMI [13].

5.3 Configuration Files and Compilation with Ant

Because the credential-creation step is complex, involving program compilation and creation of web service artifacts, 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, we measure the configuration files and include them in V_{arch}.

Program compilation can also 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 our prototype, we have to deal with the same issues, so reusing the Ant build file seems sensible. However, Ant is a powerful tool and it would be possible to write a malicious build file to avoid verification. In order to stop this from happening, the build file must be measured into the quote and included in V_{arch}. It must also be checked by the requester, along with all the libraries and files it references. In our prototype this is done manually.

Another problem arises from runtime configuration. Most applications are designed to work differently at runtime depending on how they are configured. If any of the security properties of a service can be violated by a change of settings file, then they will (correctly) not pass the verification step of our system. The same is true for a service which depends upon a running database or human input. Essentially, any system that can be made to violate its specification *after* compilation will be a problem. The alternative method discussed in Section 7 does not have this limitation.

6 Evaluation

6.1 Benefits of Trustable Remote Verification

In our proposed system 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 1, we gain the knowledge that the method `makeTransfer` will at least revert back to our previous state rather than fail in an unknown way. We also know that the error log is guaranteed to keep track of any failures. In terms of security, we could use assertions about information flow to be sure of confidentiality. JML has been used before for security properties[14].

JML's ability to express useful properties (and ESC/Java2's ability to check them) is a significant consideration when evaluating the prototype. There are properties that cannot be expressed or checked, and the properties that the service provider has asserted may not be important to the requester. Because ESC/Java2 requires a significant amount of supporting JML, time must be spent annotating the source code for each particular property. However, this time can be justified by considering some of the other benefits of Design by Contract, such as improving documentation and overall code robustness. Furthermore, we must also assume that the requester will correctly understand the given properties, and not misinterpret a set of conditions or invariants which may be complex. More research is needed to find out where Trustable Remote Verification would be most beneficial, but it is likely to be services which have simple properties to assert, such as the conditions in which an exception will be thrown, or the correct implementation of simple mathematical functions.

No additional third party is required to create the service credentials, beyond a Privacy CA which may exist already. However, as noted in Section 5.2, an integrity measurement infrastructure will be needed for verifying the trusted software components, such as TC and TV. This may require a trusted third party. 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 authenticated 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 to 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.

6.2 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, so we will now assess ours. We rely upon 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 to software attack, and hardware attacks are non-trivial. They are therefore unlikely to be the weakest point in the system.

The verification environment also 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, we must already trust the compilers we use, 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. We offer no solution to this problem, but expect that 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 likely to happen, as the amount of code running which has not been formally checked greatly outweighs the small amount that has. It includes the service middleware, operating system, and any libraries that the service relies upon. This 'middleware problem' has been discussed in a grid scenario by Cooper and Martin[15], and solutions involving virtualization sound promising. This problem is true of almost all trusted computing systems involving legacy applications, and ours is no exception.

Overall, TRV is clearly 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 on verification instead.

6.3 Related Work

Haldar et al. [16] introduce Semantic Remote Attestation, a technique for verifying the remote behaviour of a platform. They use a Trusted Virtual Machine (TVM) to attest to high-level properties of the running code. The presence of the TVM is attested first using normal methods, then the TVM can report on properties such as class hierarchies, Java VM security constraints and runtime dynamic state. Arbitrary properties can also be tested by requesting the TVM accept and run code written by the requester. This is a powerful technique, offering remote users the ability to make better trust decisions. However, the focus of this work is on dynamic feedback and testing rather than showing formal properties. We feel that our work compliments this by proving different semantic properties. Our approach is also significantly different, not relying on a runtime virtual machine, thus removing any potential performance issues.

Yoshihama et al. [17] and Monetoh et al. [13] have created and extended a web service attestation architecture called WS-Attestation. It uses a third party Validation Service (VS), which maintains an integrity database, linking software hash values to known application identities. The VS also has access to a vulnerability database. The idea is that the attesting platform reports its integrity measurement to the requester along with a certificate issued by the VS. This certificate can state a more easily interpreted property, such as the number of known vulnerabilities. However, this system is best suited to verifying the integrity of the OS and middleware of a platform, rather than of a custom application, which is unlikely to have any entries in the vulnerability database. It is also not clear how properties beyond vulnerability counts might be established.

Betin-Can et al. [18] use a design for verification approach to verify services. They introduce the 'Peer Controller Pattern' which separates out the message exchange from the logic, simplifying the verification process. An explicit behavioural interface is also generated. Assertions that are known to hold in individual services are then combined and the behaviour of the whole system can be checked with regard to synchronizability. Individual implementations are considered to conform with their interfaces if their call-sequences are acceptable to its state machine. This is verified using the JavaPathFinder model checker. It seems an excellent approach when considering concurrency issues, but does depend on all source code being available to the verifier, with trust in remote parties being less of an issue.

Tsai et al. [19] describe a framework ('WebStrar') for web service assurance. Services are registered and a series of tests are performed on it. Each service has a specification written in OWL-S 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 issues. Testing is not appropriate in a situation where the service operates on live data, and it is also not clear whether the tested services have any obligation to re-register in the case of a change to their implementation.

7 Observations and Future Work

7.1 Runtime Checking and Compile-Time Translation

A similar but alternative approach to the one taken in our prototype is to use the JML compiler[9], `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 we no longer need to perform static analysis. This was one of the issues identified in Section 6.2. As a result, runtime components such as configuration files and databases can be accessed without breaking the assertions. However, the major 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. However, despite these limitations it may be worth investigating in the future. This approach is similar to earlier work by Haldar et al. [16].

7.2 Multiple Verifications and the New Chain of Trust

One useful property of our system is that the credential-creation process is entirely separate from the runtime attestation. As a result, we can extend our prototype to offer multiple, potentially independent verifications and certificates for 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.

We can go even further, and let multiple organisations 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. Figure 3 is no longer as big a problem, as the top chain can be put in parallel with several others. This might be useful for high-assurance systems, such as e-voting.

7.3 Using a Dynamic Root Of Trust

One way in which we could optimise our proposed TRV system, with respect of the number of integrity measurements, would be to use a dynamic root of trust. Rather than measuring every part of the boot process, including the BIOS, we could use either AMD's Secure Virtual Machine (SVM) architecture or Intel's Trusted Execution Technology (TXT) to dynamically load and measure ('late launch') a trustworthy operating system. McCune et al. [20] provide a good summary of these technologies. This would reduce the number of measurements that the service requester needs to verify, and therefore make the overall system trustworthiness easier to quantify. With the recent development of a suitable bootloader[21] this would be a sensible future improvement.

7.4 Verifying Multiple Services

We have not considered verifying web services which themselves contact other services. This is quite a common scenario, and a significant limitation of our 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.

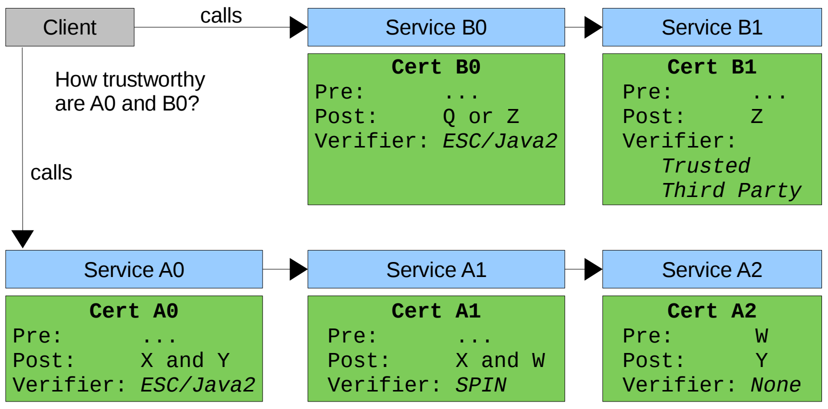

Fig. 4. The challenge of verifying service workflows

Presenting multiple service certificates presents another set of problems. Firstly, what happens if one of the sub services is not considered trustworthy? This could be for a number of reasons, for example the verification tool might be out of date, or the AIK certificate could have expired. The implications of this may affect all or part of the overall combined service. Secondly, do we still trust a system with so many trusted components? If a TPM was successfully attacked on any of the sub services it might compromise the whole process. Finally, it is possible that we will be contacting services which do not use this certification system. They may have a different trust mechanism, or none at all. How to weight the trustworthiness of this service is an open question.

8 Conclusion

We have introduced the idea of Trustable Remote Verification, a technique which allows a service provider to verify its own software and create a trustworthy guarantee of the result. The main advantages are that no new third parties are needed, and providers can easily create a new verification result whenever necessary. Furthermore, one verification can be used as a credential for many platforms. We have also identified the key shortcomings and limitations to our work, including the large amount of necessarily trusted code, and the need for an annotated, verifiable service. However, our prototype implementation uncovered few additional issues, and we feel justified in persuing these ideas further. Overall, this technique has great potential, and could be a viable way of adding assurance to Service Oriented Architectures. Future work will concentrate on finding where the best applications of this idea are, what properties it can guarantee, and whether architectural approaches can be used to reduce the trusted computing base of verified services.

Acknowledgements

Special thanks go to Andrew Martin and Jun Ho Huh for several helpful discussions while writing this paper and developing the concepts. I am also grateful to Joe Loughry, Andrew Cooper, Shamal Faily and Cornelius Namiluko for their comments. Finally, I thank the reviewers for their useful and positive feedback.

References

1. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1. Technical report, W3C (March 2001) `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.
2. The Trusted Computing Group: TCG Specification Architecture Overview, Revision 1.4. `https://www.trustedcomputinggroup.org/groups/TCG_1_4_Architecture_Overview.pdf` (August 2007)
3. The Trusted Computing Group: TCG Glossary of Technical Terms. `https://www.trustedcomputinggroup.org/groups/glossary/` (2008)
4. Poritz, J.A.: Trust[ed | in] Computing, Signed Code and the Heat Death of the Internet. In: SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing, New York, NY, USA, ACM Press (2006) 1855–1859
5. Sadeghi, A.R., Stüble, C.: Property-based Attestation for Computing Platforms: Caring About Properties, Not Mechanisms. In: NSPW '04: Proceedings of the 2004 Workshop on New Security Paradigms, New York, NY, USA, ACM Press (2004) 67–77
6. Papazoglou, M.P., Dubray, J.j.: A Survey of Web Service Technologies. Technical Report DIT-04-058, Informatica e Telecomunicazioni, University of Trento (June 2004)
7. The W3C: Simple Object Access Protocol (SOAP). `http://www.w3.org/TR/soap/` (April 2007)

8. Meyer, B.: Design by Contract: Building Reliable Software. In: Object-Oriented Software Construction. Prentice Hall (1997) 331–341
9. Leavens, G., Cheon, Y.: Design by Contract with JML. `http://citeseer.ist.psu.edu/leavens04design.html` (2003)
10. Cok, D.R., Kiniry, J.: ESC/Java2: Uniting ESC/Java and JML. In Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T., eds.: CASSIS. Volume 3362 of Lecture Notes in Computer Science., Springer (2004) 108–128
11. Necula, G.: Proof-Carrying Code. Website (July 2002) `http://raw.cs.berkeley.edu/pcc.html`.
12. Jaeger, T., Sailer, R., Shankar, U.: PRIMA: Policy-Reduced Integrity Measurement Architecture. In: SACMAT. (2006) 19–28
13. Munetoh, S., Nakamura, M., Yoshihama, S., Kudo, M.: Integrity Management Infrastructure for Trusted Computing. IEICE Transactions on Information and Systems **E91-D**(5) (2008) 1242–1251
14. Pavlova, M., Barthe, G., Burdy, L., Huisman, M., Lanet, J.L.: Enforcing High-Level Security Properties for Applets (2004)
15. Cooper, A., Martin, A.: Towards a secure, tamper-proof grid platform. CCGRID 06. Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. **1** (May 2006) 8
16. Haldar, V., Chandra, D., Franz, M.: Semantic Remote Attestation - Virtual Machine Directed Approach to Trusted Computing. In: Virtual Machine Research and Technology Symposium, USENIX (2004) 29–41
17. Yoshihama, S., Ebringer, T., Nakamura, M., Munetoh, S., Maruyama, H.: WS-attestation: efficient and fine-grained remote attestation on Web services. ICWS 2005. Proceedings. 2005 IEEE International Conference on Web Services (July 2005) 743–750
18. Betin-Can, A., Bultan, T.: Verifiable Web services with Hierarchical Interfaces. ICWS 2005. Proceedings. 2005 IEEE International Conference on Web Services (July 2005) 85–94 vol.1
19. Tsai, W., Wei, X., Chen, Y., Xiao, B., Paul, R., Huang, H.: Developing and assuring trustworthy Web services. Autonomous Decentralized Systems, 2005. ISADS 2005. Proceedings (April 2005) 43–50
20. McCune, J.M., Parno, B.J., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: an execution infrastructure for tcb minimization. In: Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, New York, NY, USA, ACM (2008) 315–328
21. Wei, J., Cihula, J., Wang, S.: Trusted Boot Sourceforge Project Website (2008) `http://sourceforge.net/projects/tboot/`.