SAConf: Semantic Attestation of Software Configurations[*]

Hua Wang, Yao Guo, and Xiangqun Chen

Key Laboratory of High Confidence Software Technologies (Ministry of Education)
Institute of Software, School of EECS, Peking University, Beijing, China
{wanghua04,yaoguo,cherry}@sei.pku.edu.cn

Abstract. Remote attestation is one of the key functionalities provided by trusted platforms. Most current attestation approaches are based on cryptographic hash functions, which are appropriate to attest to relatively stable objects such as executables. However, they can not effectively deal with software configurations that could have many (or even infinite) trusted variants and could also be modified at run-time. This paper proposes SAConf, a novel semantic attestation approach to attesting to software configurations. SAConf uses a list of constraints to represent the challenger's trust policies, and verifies configurations based on semantic checks against the constraints, according to the semantic meanings of configurations rather than their hashes. An on-request measurement strategy is also added as a complement to the on-load strategy in order to capture potential modifications to configurations during execution. We implemented a prototype of SAConf and evaluations show that it could reduce the storage overhead from *exponential* to *linear* compared to hash-based approaches.

1 Introduction

In a distributed environment involving multiple platforms, the platforms could be owned and managed by different entities who might not trust each other. Platforms could also be compromised and running malicious code. Thus it is very important that a platform (*challenger*) is able to verify the software trust state of another platform (*attestor*).

Configuration is an important factor affecting software trust. Many programs can be tuned to behave in very different ways through user-specified configurations. Usually a program is not trusted if its configuration does not comply with the challenger's trust policy, even if its executable is launched correctly without being tampered with. Therefore attesting to configurations should be included in software attestation as well.

[*] This research is supported by the National High Technology 863 Program of China under Grant No. 2007AA01Z462 and 2008AA01Z133, the National Basic Research Program of China (973) under Grant No. 2009CB320703, and the Science Fund for Creative Research Groups of China under Grant No. 60821003.

The Trusted Computing Group (TCG) has developed a bottom-up measuring model and a hardware-based integrity-proving mechanism [21], based on which some attestation approaches have been proposed, such as TPod [8] and IMA [14]. These approaches are believed to be capable of reporting trust states more reliably than pure software approaches. Existing TCG-style approaches do not distinguish between executables and configurations. Both are proved using cryptographic hash functions and with the on-load measure strategy. That is, the attestor measures executables and configurations by computing their hashes at load time, and reports their hashes to the challenger during attestation. The challenger verifies these hashes by comparing them with pre-stored, trusted hashes.

Although TCG-style approaches may be suitable for executables, they do not work well for configurations due to the following two reasons. 1) When dealing with configurations, the hash space could explode very easily. Unlike executables that do not have many trusted variants, the number of trusted configurations could be extremely large or even infinite, for example, when considering a configuration entry that accepts a float number within a given range. Since with TCG-style approaches each configuration is denoted by a unique hash, it is sometimes impractical or impossible to deal with all trusted hashes, such as storing all of them. 2) The on-load measurement strategy can not always reflect the latest configuration. While executables usually do not change after being loaded, configurations of programs, such as *Firefox*, could be modified on the fly. In such cases the measurement performed at load time can not accurately indicate the trust state.

Some approaches have been proposed to mitigate the problem of hash explosion. Virtual machine (VM) based approaches such as Terra [3] separate trusted VMs and normal VMs, and only verify the hashes of programs running in the trusted VMs. Property-based attestation [12, 11, 1] introduces a trusted third party to examine the attestor's hashes and returns its properties rather than hashes to the challenger. PRIMA [5] only attests to programs having information flow to trusted objects. These approaches, however, are still based on hash functions and focused mainly on executables.

The main limitation of hash-based approaches is that they do not take the internal structural information of configurations into account because this information can not be carried by hash values. Hence semantic checks such as range comparison and pattern matching can not be performed, thus the trust policy can only be represented by enumerating all trusted hashes.

To address this problem, we propose a new attestation approach, called Semantic Attestation of Configuration (SAConf), which represents the challenger's trust policy in a semantic way (i.e. using a group of constraints), and verifies configurations against these constraints according to the internal contents of configurations rather than their hashes. The verification is based on semantic checks, so the challenger could use a small list of constraints to match a large number of configurations. To protect the attestor's privacy, the configuration is not sent to the challenger. Instead, the challenger sends its trust policy to the attestor, and the attestor is responsible for verifying whether its configuration complies the policy and reporting the result. To assure the challenger of the genuineness of the attestation result, the attestor also proves that the measuring and verifying code is executed correctly.

We add an on-request measurement strategy to SAConf as a complement to the on-load strategy. With this strategy the attestor performs configuration measurement each time when receiving an attestation request from the challenger, so that the measurement result always reflects the current state at the moment when the request is processed. We implemented a prototype of SAConf to demonstrate its feasibility, in which we develop an example trust policy representation scheme for entry-based configurations, reducing the storage cost from exponential to linear. The time cost may increase with the on-request strategy, but experiment results show it is trivial.

The rest of this paper is organized as follows. Section 2 describes the design of SAConf. Section 3 describes the implementation and evaluation of the prototype. Section 4 discusses several additional advantages of SAConf. The related work is discussed in section 5. Finally we draw a conclusion and present our future work in section 6.

2 Semantic Attestation

TCG-style attestation approaches are based on cryptographic hash functions, requiring the challenger to store one hash for each trusted file. For executables, this method works well, since the number of trusted variants of an executable is not very big, usually including the original version and a number of patched versions. However, the number of trusted configurations could be extremely large, sometimes even infinite. For example, in terms of entry-based configurations consisting of <*entry*, *value*> pairs, a challenger may accept more than one value as trusted for some configuration entries. The values of these entries could be combined to produce a tremendous amount of trusted configurations. Worse still, some entries may even have unlimited choices. For example, it is not odd that the challenger does not care about the name of the user who runs Apache's *httpd*, as long as its privileges are properly set. In this case the *User* entry of *httpd*'s configuration could be assigned arbitrary names, leading to actually countless trusted configurations. Consequently, it is often impractical or even impossible for the challenger to store hashes of all trusted configurations.

To attest to configurations which have internal structures and can be parsed according to their syntax, a better way is to read their contents and perform semantic checks on them to determine whether they comply with the trust policy. A trust policy usually consists of a group of constraints, such as "*the server at least supports hmac-sha1 algorithm*", which must be satisfied by trusted configurations. We can represent the constraints in a formal way, so that semantic checks can be done automatically.

Semantic checks require SAConf to understand syntax of configurations. Hence some operations definitely depend on the configuration syntax of programs to be attested, including representing the challenger's trust policy, and measuring and checking the configuration. Configurations of different programs could be in very diverse forms, such as entry-based, rule-based and command-based configurations. It is hard to develop a common solution for all programs. Alternatively, we develop a flexible framework in which syntax-dependent operations are encapsulated in customizable and replaceable components.

Semantic checks also require that the party who performs the checks knows both the configuration and the trust policy. There are several candidates. A straightforward method is that the attestor sends the entire configuration to the challenger, which in turn checks the configuration against its policy. This approach, however, exposes the attestor's privacy to the challenger. A more sophisticated method is that both the configuration and the trust policy are sent to a reliable third party, which performs the check on behalf of the challenger. But this method requires the presence of an extra third party that is not always available. The method adopted by SAConf is that the challenger sends its policy to the attestor and the latter is responsible for performing the check. This method eliminates the above two deficiencies.

Because the configuration measurement and checks are all done in the attestor side, the attestor should provide necessary proof to convince the challenger of the genuineness of the attestation result. The facts to be proved include that the components of SAConf are not compromised and the trust policy and attestation result are not tampered with. To do so SAConf provides the attestor with a proving mechanism built on TCG's technology.

3 Design of SAConf

This section describes the design of SAConf, including the framework and the attestation process, as well as the mechanism for proving the genuineness of attestation results.

3.1 Framework of SAConf

The framework of SAConf is depicted in Figure 1, where syntax-dependent components are denoted by grey boxes, while syntax-independent components are denoted by white boxes.

The challenger needs to store its *trust policy* and send it to the attestor during attestation. Hence SAConf should provide the challenger with schemes to represent its policies. It is impractical to design a common scheme for all programs, but we could improve the flexibility of schemes so that they can be shared by a group of programs with similar configuration syntax. In next section we will present an example scheme which can be used for most entry-based configurations.

The *Attestation Server* (*AttServ*) runs as a daemon in the attestor. It is responsible for handling the interaction with the challenger and coordinating the activities of other components within the attestor.

The *Measurement Engine* (*MEngine*) is responsible for measuring configurations. Programs with distinct configuration syntax usually require customized

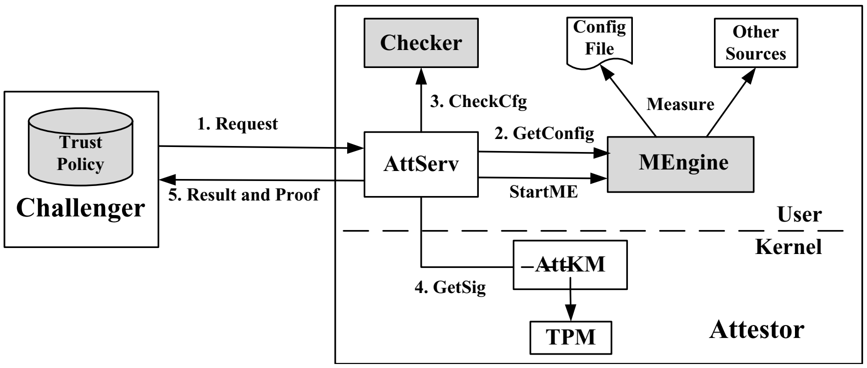

Fig. 1. SACon Framework

MEngines. The relationship between programs and MEngines is registered in a dedicated file.

The *Checker* performs semantic checks, examining whether the configuration measured by the MEngine satisfies the trust policy received from the challenger. The checking method depends on the policy representation scheme.

The *Attestation Kernel Module* (*AttKM*) is a kernel module used to invoke TPM to generate signatures to prove the genuineness of the attestation result. It also monitors execution of programs and notifies the AttServ of this information.

3.2 Attestation Process

The configuration attestation is accomplished through the collaboration of the components. The numbers in Figure 1 stand for the temporal order.

In terms of configuration measurement, most existing attestation approaches use the on-load strategy, i.e. measuring the configuration when it is loaded. We propose adding n on-request strategy, i.e. measuring the configuration when an attestation request is processed. SAConf supports both strategies. The former is suitable for programs whose configurations do not change on the fly, such as *sshd* and *httpd*; while, for programs whose configurations may be modified dynamically, e.g. *Firefox*, the latter is desired. Which strategy is used can be configured for each program.

Measurement is performed by MEngines launched by the AttServ. If a program is configured to use the on-load strategy, its MEngine is launched when the program is executed. Monitoring execution of programs is done by the AttKM, which hooks the kernel function used to execute programs, e.g. *do_execve* in Linux, and signals the AttServ if execution is detected. If the on-request strategy is used, the MEngine is launched after the AttServ receives a request from the challenger. In both cases the AttServ calculates the hashes of MEngines when launching them, in order to prove their integrity.

If the on-load strategy is used, the measurement result, as well as the hash of the MEngine, needs to be stored and used for attestation requests arriving later. We use a dedicated Platform Configuration Register (PCR) [21] to protect both of them. While, with the on-request strategy, the result and the hash are consumed immediately after measurement and only for the current request, so it is not necessary to store them.

The attestation process varies slightly when using different strategies. The detail of each step is described as follows.

1. The attestation begins with the challenger sending a request to the AttServ, which contains the identity of the program to be attested and the trust policy for it, as well as some assistant data such as *nonce*, a random number used to defend against replay attacks.
2. Upon receipt of the request, the AttServ retrieves the program's identity and finds out which measurement strategy is configured for the program. If the on-load strategy is used, the AttServ fetches the configuration measured at load time, as well as the hash of the MEngine, and examines their integrity according to the PCR. Otherwise the AttServ looks up the MEngine registered for the program, and launches it to measure the current configuration, with the hash of the MEngine being computed.
3. The AttServ then launches a proper Checker, passing the trust policy and the configuration to it. The Checker examines whether the policy is satisfied by the configuration based on semantic checks. Like the MEngine, the Checker's hash is also calculated, used to prove its integrity.
4. The AttServ requests the TPM to generate a signature through the system call provided by the AttKM, used as the evidence of the genuineness of the attestation result. The computation of the signature involves the attestation result, the trust policy, and the hashes of SAConf's components.
5. The result, the hashes and the signature are sent back to the challenger. The challenger first verifies the integrity of the trust policy, the result and the hashes according to the signature. Then it verifies according to the hashes whether the SAConf's components are valid and launched correctly. If so, the challenger is assured that the result denotes the actual trust state of the program's configuration.

The attested program does not participate in the attestation process. There is also no synchronization requirement between the components of SAConf and the program. So applying SAConf does not need to modify existing programs.

3.3 Proving the Genuineness of Attestation Results

SAConf provides the attestor with a proving mechanism to vouch for the genuineness of the attestation result. This mechanism is built upon TCG's technology, based on the assumption that a traditional TCG-style attestation approach, such as [8, 9, 14], has been implemented in the attestor side.

TCG-style attestation approaches measure all loaded programs and modules, and the measurement results are categorized and stored in separate sequences, such as the sequences for the kernel and for applications, with corresponding

PCRs being extended. These sequences, as well as PCRs, are sent to the challenger during attestation. The challenger verifies the integrity of these sequences by calculating their hashes and comparing them with corresponding PCRs.

We use the traditional approach to prove low-level software from BIOS up to the kernel, including the AttKM. But the rest of SAConf's components, which are user-mode programs, are not proved by the traditional approach, due to the following two reasons. 1) MEngines and Checkers should be not only legal but also proper. Here the term "legal" denotes that a program has been certified by a trusted third party, while "proper" denotes that a program works correctly in a certain situation. A legal program may be not proper if used in a wrong place. For example, the MEngines for *sshd* and *httpd* are all legal, but using the former to measure *httpd* is obviously not proper. Traditional approaches can not detect the improper use of legal programs. 2) When verifying configurations, the challenger usually only concerns SAConf's components. But with traditional approaches, the challenger has to process the whole sequence containing the hashes of all loaded applications. The cost of this operation is high, especially when the system has been running for a long period and a lot of loads have occurred.

We propose a customized, more lightweight mechanism to prove the user-mode components. For the AttKM has been proved by the traditional approach, we use it to prove the AttServ, which in turn is used to prove Checkers and MEngines. The AttServ measures Checkers and MEngines when launching them. The AttServ itself is measured by the AttKM when it invokes the system call provided by the AttKM to generate signatures. The measurement results, i.e. hashes, are sent back to the challenger along with the attestation result, so the challenger can tell which MEngine and Checker are used.

The AttKM invokes the TPM to generate a signature to prove the hashes of SAConf's components. Because only the kernel is allowed to access the TPM, this signature can not be forged by malicious applications. Besides, in order to prove the integrity of the trust policy and the attestation result, their hashes are also included in the computation of the signature.

Finally, what the challenger receives from the attestor is

$$< result, hs, sig >$$

where *hs* and *sig* are the proof. *hs* is the hashes of the Checker, the MEngine and the AttServ. *sig* is the signature generated by the TPM whose value is

$$sig = S_K(H(result), H(tp), H(hs), nonce)$$

where *S()* and *K* are the signing function and key used by the TPM; *H()* is a hash function such as SHA1; *tp* is the challenger's trust policy; and *nonce* is a random number generated by the challenger to prevent replay attacks.

When receiving the package, the challenger retrieves the attestation result, *hs* and *sig*. It also knows *tp*, *nonce* and the public part of *K*. Then it verifies the genuineness and freshness of the package by examining the signature and *nonce*, followed by verifying the hashes of SAConf's components. If all verification is passed, the challenger believes that the attestation result reflects the actual trust state of the program's configuration.

4 Implementation

We implemented a prototype of SAConf in a Dell OptiPlex 620 equipped with a TPM chip, running Linux 2.6.20. This section describes its implementation and evaluation.

4.1 Representation Scheme for Entry-Based Configurations

Although developing a thoroughly common trust policy representation scheme for all programs is infeasible, we could develop flexible schemes that can be used for a large number of programs. The entry-based configuration is one of the most widely used configuration forms. We develop an example scheme that can be used for most of this kind of configurations.

For entry-based configurations, a challenger's trust policy are typically some logical conditions each of which must be satisfied by the values of configuration entries. For example, a challenger requires the value of an entry must be greater than a constant. Straightforwardly, we use a set of boolean expressions to represent these conditions. Each expression involves one or several configuration entries. If their values make the expression true, the expression is said to be satisfied. If all expressions are satisfied, the configuration of the program is considered to be compliant with the challenger's policy.

The boolean expression is somewhat similar to that of high-level languages. Most of often-used operators are supported in our scheme. For entries with numeric values, all arithmetic, relational and logical operators are supported. For entries with string values, "==" and "!=" are supported, as well as some frequently used string functions such as *strcmp*, *strlen* and *strstr*. Entry values are referred to by the *$()* operator. *$(entryid)* will be replaced by the value of the entry specified by *entryid* when the expression is evaluated.

In addition, we add two extensions to enhance the representation capability of our scheme. The first extension is to support set operators. In some cases it is convenient for the challenger to treat some entries (e.g. lists) as sets. For example, the challenger may require that its ID is in the trusted ID list. This can be represented by a *belong* operator easily. Supported set operators are listed in Table 1. Two special sets are defined. The empty set is denoted by Φ and the universe is denoted by Ψ. The second extension is to support regular expressions. Regular expressions can be used to specify entry IDs in the *$()* operator, or describe patterns that the values of trusted entries must match.

An example trust policy for *sshd* represented by our scheme is shown in Figure 2. Its meaning is straightforward.

Our scheme enables semantic checks such as range comparison and regular expression, which are not supported by hash based approaches. With this scheme

Table 1. Supported Set Operators

Operator	Syntax	Description
set	set(d,L)	This operator is used to convert a list to a set. L is the string that denotes the list, and d is the delimiter that separates elements in the string.
==	S_1 == S_2	If S_1 contains the same elements as S_2, return *true*; otherwise return *false*.
!=	S_1 != S_2	Return the reverse result of ”==”.
belong	e belong S	If element e belongs to S, return *true*; otherwise return *false*.
incl	S_1 incl S_2	This is the inclusion operator. If S_1 includes S_2, return *true*; otherwise return *false*.
union	S_1 union S_2	This operator returns the union of S_1 and S_2.
inters	S_1 inters S_2	This operator returns the intersection of S_1 and S_2.
diff	S_1 diff S_2	This operator returns the difference of S_1 and S_2.

```
[sshd, f8e3e1cd58fdb8434497c0c9ca783bc3cf6a38b3]
// The supported protocol version must be included in set {1,2}
#1 set(,,"1,2") incl set(,,$(Protocol))

// root is not allowed to log in through ssh
#2 !("root" belong set( ,$(AllowUsers))) || ("root" belong set( ,$(DenyUsers))) \
|| ($(PermitRootLogin) == "no")

// The intersection of AllowUsers and DenyUsers should be a empty set
#3 (set( ,$(AllowUsers)) inters set( ,$(DenyUsers))) == Φ

// the cvs group is allowed to log in
#4 "cvs" belong set( ,$(AllowGroup))

// expressions for password authentication
#5 $(PasswordAuthentication) == "yes"
#6 $(PermitEmptyPasswords) == "no"
#7 $(UsePAM) == "yes"
#8 $(MaxAuthTries) <= 6
#9 $(MaxStartups) > 5 && $ < 10
```

Fig. 2. An Example Trust Policy for *sshd*

the challenger can use a group of boolean expressions to match a large number of configurations, so its storage overhead decreases significantly.

4.2 Evaluation

In this subsection we will evaluate the storage and time cost of SAConf respectively.

Storage Cost. Our trust policy representation scheme reduces the storage complexity for the challenger from *exponential* to *linear*.

With hash-based approaches the challenger needs to store one hash for each trusted configuration. The number of trusted configurations can be computed approximately as follows. Suppose there are n entries in the configuration and

Table 2. Time Cost

	Measuring	Checking	Reporting	Responding
SAConf(On-request)	544μs	9μs	932751μs	933304μs
SAConf(On-load)	24166μs	135μs	931773μs	931908μs
IMA	23394μs	-	927168μs	927168μs

the number of trusted values for the ith entry is c_i. Then the number of trusted configurations is $\prod_{i=1}^{n} c_i$. So the storage cost grows *exponentially* with the number of entries having multiple trusted values. Here we do not take factors such as comments and the order of entries into account, otherwise the cost will be infinite.

While with our scheme, the challenger only needs to store some boolean expressions. So the storage cost grows *linearly* with the number of the constraints, which is usually less than the number of entries.

Time Cost. We measured the time cost of SAConf and compared it with that of IMA. The result is shown in Table 2. The operations done in the attestor side can be divided into three stages: measuring the configuration, checking the configuration, and reporting the result. Their cost is listed in corresponding columns.

Each stage is comprised of several operations, some of which are accomplished by invoking TPM commands. Because of the limited computational ability of TPM, the time spent by these commands accounts for the majority of the cost.

When using the on-load measurement strategy, the measurement is only performed at load time, but SAConf needs to extend PCRs for each measurement, which is similar to IMA, making its measurement cost close to that of IMA. When using the on-request strategy, the measurement is only used for the current attestation request. SAConf does not need to store the result and extend PCRs to protect it, so the cost is much lower.

The checking stage is only necessary for SAConf. With the on-load strategy, after retrieving the previously measured and stored result, SAConf still needs to get the specific PCR and verify the result accordingly; while this operation is not necessary when using the on-request strategy. Therefore the cost of the latter is much lower than that of the former.

With respect to the reporting stage, the cost of SAConf and IMA is close. In the reporting stage, SAConf needs to invoke the TPM to generate a signature as evidence, while IMA needs to get a signed PCR from the TPM. Both need the signing operation of the TPM, which contributes to more than 99% of the cost of this stage.

The last column shows the responding time, namely the time from an attestation request being received to the final result for the request being returned. For SAConf with the on-request strategy, the responding time includes the time spent in all stages; for SAConf with the on-load strategy, it includes the checking and reporting stages; while for IMA, it only contains the reporting stages. But from the result we can see that there is no big difference between their responding time.

Consequently, although some operations of SAConf, such as measuring for each attestation request, parsing configuration files and checking semantically, may cause extra time cost, it has little compact on the total performance because the majority of time overhead is spent by the TPM.

5 Related Work

Establishing trust among platforms is an important requirement in distributed environment. This is usually done by the attestation technologies. There have already been a lot of attestation approaches proposed.

Some approaches are based on secure hardware. TCG has developed TPM [20], a secure chip that has been shipped with many platforms. Based on TPM TCG proposed a hash-based integrity-proving mechanism [21]. The hashes of files, including executables and configuration files, are computed at the load time, and reported to the remote platform later as the trust evidence.

TCG's proving mechanism has been widely adopted. TPod [8] implements extensions to the grub bootloader to measure the sequence of code loads that bring up the operating system, and it stores these measurements in the TPM to protect them from tampering by software. The TPM can create signed messages that enable a remote party to verify the code loads measured by TPod. IMA [14] steps further by extending integrity measurement and verification up to the application level. Sailer et al. limit clients' access to the corporate network according to their integrity property [13].

Some researches attempt to make improvements to TCG's mechanism. Property-based attestation [12, 11, 1] concerns with the platform's property, such as security level, rather than the hashes of loaded files. A trusted third party is introduced to map hashes to platform's properties. Through the property-based attestation, [11] intends to protect the attestor's privacy. Instead, in SACon we use the privacy policy mechanism to do so.

Semantic remote attestation [4] attempts to attest to the program's behavior, rather than the integrity of its executable and configuration file. The approach is based on language-based trusted virtual machines (VM). It utilizes the high-level semantic information contained in the portable code to deduce the program's behavior. Although we all use the word "semantic", it has different meanings. In semantic remote attestation it means the information in executables, while in SACon it denotes the information in configuration files.

Attesting to executables will also encounter the problem of multiple versions, though this problem is not so serious as that of configuration files. Some researches have identified and tried to solve this problem. Terra [3] uses a VM based approach, which provides trusted VMs and normal VMs simultaneously, running high security-level and low-security level applications separately. The challenger only needs to verify the software stack in trusted VMs, so the hashes that the challenger needs to store are reduced. PRIMA [5] extends the IMA by coupling IMA to SELinux policy [7]. The number of measurement targets is reduced to those that have information flows to trusted objects. BIND [19] allows programmers to mark out the critical code region through an attestation annotation mechanism. Only the hashes of the critical regions are computed and reported. These regions are more stable than other code and data regions. However, all these approaches focus on the attestation of executables. None of them can solve the problem of configurations effectively.

The attestation result can only prove the transient trust state, but not persistent state. BIND [19] attempts to prolong the trust state by moving attested code into a sand-boxing to protect its execution. However BIND only concerns with code, but not configurations. SACon mitigates this problem by providing the challenger with the capability of requesting the attestation at any moment. The challenger can send requests periodically or at random time.

In addition to hardware based approaches, there have also been some pure software approaches proposed. Genuinity [6] explores the problem of detecting the difference between a simulator-based computer and an actual computer. Genuinity relies on the premise that the program execution based on simulator is bound to be slower than that based on the actual computer. The execution time of a specific function that computes the checksum of memory is used as the evidence to make the attestation. However, Shankar et al. shows that side effects are not enough to make the attestation [18]. Pioneer [15] also relies on the execution time to perform the attestation. A deliberately designed verification function and a challenger-response protocol are used to establish the dynamic root of trust, which can ensure the succeeding code execution. TEAS [2] sends a randomly selected code segment, rather than a fixed function, to the remote platform, and determines the genuineness of the remote platform according to the returned result and the consumed time.

SWATT [16] is a technique proposed to perform attestation on embedded device with simple CPU architecture. It uses a well-constructed verification function so that any attempt to tamper with it will increase the running time. There are still some other attestation approaches proposed for embedded systems [10, 17, 22].

6 Conclusion and Future Work

Trust attestation of a software system must take its configuration into account. Existing attestation approaches can not attest to the software configuration effectively because of high storage overhead and inability to prove the latest states of configurations that can be modified dynamically.

In this paper we propose SAConf, a semantic attestation approach, to overcome these problems. The key contribution of SAConf is that it integrates semantic checks with TCG's hardware-based proving mechanism. The genuineness of the checks is proved by a TPM-based mechanism. The challenger's trust policies are also represented semantically, reducing the space complexity from exponential to linear. Besides, we introduce an on-request measurement strategy, which can accurately reflect the trust state at the moment when the request is processed.

We demonstrate the feasibility of SAConf by implementing a prototype. Experiments show that SAConf could reduce the storage overhead significantly compared to state-of-the-art approaches, only with a very little time cost increase.

Currently we have developed a trust policy representation scheme for entry-based configurations. Schemes for other configuration forms, such as rule-based and command-based configurations, still need to be developed, and could be very different from this one. In future we will study these configuration forms and develop proper schemes for them.

References

[1] Chen, L., Landfermann, R., Lohr, H., Rohe, M., Sadeghi, A.-R., Stable, C.: A Protocol for Property-Based Attestation. In: The 1st ACM Workshop on Scalable Trusted Computing, Alexandria, Virginia, USA, pp. 7–16. ACM, New York (2006)

[2] Garay, J.A., Huelsbergen, L.: Software Integrity Protection Using Timed Executable Agents. In: The 2006 ACM Symposium on Information, Computer and Communications Security, Taipei, Taiwan, pp. 189–200 (2006)

[3] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: A Virtual Machine-Based Platform for Trusted Computing. In: The 19th Symposium on Operating System Principles, Bolton Landing, New York, USA, pp. 193–206 (2003)

[4] Haldar, V., Chandra, D., Franz, M.: Semantic Remote Attestation - A Virtual Machine Directed Approach to Trusted Computing. In: The Third Usenix Virtual Machine Research and Technology Symposium, San Jose, CA, USA, pp. 29–41 (2004)

[5] Jaeger, T., Sailer, R., Shankar, U.: PRIMA: Policy-Reduced Integrity Measurement Architecture. In: The 11th ACM Symposium on Access Control Models and Technologies, Lake Tahoe, California, USA, pp. 19–28. ACM Press, New York (2006)

[6] Kennell, R., Jamieson, L.H.: Establishing the Genuinity of Remote Computer Systems. In: The 12th USENIX Security Symposium, Washington, DC, USA, pp. 295–308 (2003)

[7] Loscocco, P., Smalley, S.: Integrating Flexible Support for Security Policies into the Linux Operating System. In: FREENIX Track: 2001 USENIX Annual Technical Conference, Boston, Massachusetts, USA, pp. 29–42 (2001)

[8] Maruyama, H., Seliger, F., Nagaratnam, N., Ebringer, T., Munetoh, S., Yoshihama, S., Nakamura, T.: Trusted Platform on Demand. Technical Report RT0564, IBM (February 2004)

[9] Microsoft. Secure Startup - Full Volume Encryption: Technical Overview (April 2005)

[10] Park, T., Shin, K.G.: Soft Tamper-Proofing via Program Integrity Verification in Wireless Sensor Networks. IEEE Transactions on Mobile Computing 4(3), 297–309 (2005)

[11] Poritz, J., Schunter, M., Van Herreweghen, E., Waidner, M.: Property Attestation - Scalable and Privacy-friendly Security Assessment of Peer Computers. Technical Report RZ 3548, IBM Zurich Research Laboratory (October 2004)

[12] Sadeghi, A.-R., Stuble, C.: Property-based Attestation For Computing Platforms: Caring about Properties, Not Mechanisms. In: The 2004 workshop on New Security Paradigms, Nova Scotia, Canada, pp. 67–77 (2004)

[13] Sailer, R., Jaeger, T., Zhang, X., van Doorn, L.: Attestation-based Policy Enforcement for Remote Access. In: The 11th ACM Conference on Computer and Communications Security, Washington, DC, USA, pp. 308–317. ACM Press, New York (2004)
[14] Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: 13th USENIX Security Symposium, San Diego, California, pp. 223–238 (2004)
[15] Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.: Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In: Advances in Information Security, vol. 27, pp. 253–289. Springer, US (2005)
[16] Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.: SWATT: SoftWare-based ATTestation for Embedded Devices. In: The 2004 Symposium on Security and Privacy, pp. 272–282 (2004)
[17] Shaneck, M., Mahadevan, K., Kher, V., Kim, Y.: Remote Software-Based Attestation for Wireless Sensors. In: Molva, R., Tsudik, G., Westhoff, D. (eds.) ESAS 2005. LNCS, vol. 3813, pp. 27–41. Springer, Heidelberg (2005)
[18] Shankar, U., Chew, M., Tygar, J.D.: Side Effects Are Not Sufficient to Authenticate Software. In: The 13th USENIX Security Symposium, pp. 89–102 (2004)
[19] Shi, E., Perrig, A., Van Doorn, L.: BIND: A Fine-grained Attestation Service for Secure Distributed Systems. In: IEEE Symposium on Security and Privacy, pp. 154–168 (2005)
[20] TCG. TPM Main Part 1 Design Principles (March 2006)
[21] TCG. TCG Specification Architecture Overview (August 2007)
[22] Yang, Y., Wang, X., Zhu, S., Cao, G.: Distributed Software-based Attestation for Node Compromise Detection in Sensor Networks. In: The 26th IEEE International Symposium on Reliable Distributed Systems, pp. 219–228. IEEE Computer Society, Los Alamitos (2007)