A Practical Property-based Bootstrap Architecture

René Korthaus 1  
rene.korthaus@rub.de  

Ahmad-Reza Sadeghi 1  
ahmad.sadeghi@trust.rub.de  

Christian Stüble 2  
stueble@sirrix.com  

Jing Zhan 1, 3  
jingzhan_whu@yahoo.com.cn  

1 Horst Görtz Institute for IT-Security  
Ruhr-University Bochum, Germany  

2 Sirrix AG security technologies  
Germany  

3 Computer School  
Wuhan University, China  

ABSTRACT

Binary attestation, as proposed by the Trusted Computing Group (TCG), is a pragmatic approach for software integrity protection and verification. However, it has also various shortcomings that cause problems for practical deployment such as scalability, manageability and privacy: On the one hand, data bound to binary values remain inaccessible after a software update and the verifier of an attestation result has to manage a huge number of binary versions. On the other hand, the binary values reveal information on platform configuration that may be exploited maliciously.

In this paper we focus on property-based bootstrap architectures with an enhanced boot loader. Our proposal improves the previous work in a way that allows a practical and efficient integration into existing IT infrastructures. We propose a solution of the version rollback problem that, in contrast to the existing approaches, is secure even if the TPM owner of the attested platform is untrusted without requiring an interaction with a trusted third party.

Finally, we show how our architecture can be applied to secure boot mechanisms of Mobile Trusted Modules (MTM) to realize a "Property-Based Secure Boot". This is especially important for human users, since with secure boot, users can rely on the fact that a loaded system is also in a trustworthy state.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection

General Terms

Security

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

STC’09, November 13, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-788-2/09/11 ...$10.00.

Keywords

Trusted Computing (TC), Property-based Attestation, Secure Boot, Mobile Trusted Module (MTM)

1. INTRODUCTION

Binary attestation, as proposed by the Trusted Computing Group (TCG) based on the Trusted Platform Module (TPM) [22, 23, 24], is a pragmatic approach of performing software integrity protection.

Especially in the context of centrally managed IT infrastructures, as it is common in medium-sized and large enterprises, binary attestation and binary sealing can help enforcing security policies by authenticating both hardware platforms and software configurations.

However, as already pointed out by previous work (e.g., [17, 9]), the use of binary measurements (hash values over binary codes) has several disadvantages. For instance, they render sealed data inaccessible after software updates and leak platform configuration information allowing remote attackers to optimize their attacks.

In order to solve these problems, the concept of property-based attestation and sealing was introduced [17, 12]. The main idea is to use software properties instead of binary measurements for attestation and sealing, which ensures that sealed data is available after a software update as long as the new software provides the same properties. Several concrete schemes have been proposed based on different trust models and assumptions. While some solutions require changes to the current TPM, others (such as [12]) present a property-based attestation and sealing architecture that can be instantiated by the existing TCG-compliant hardware using an extended bootstrap architecture.

Contribution. In this paper we put forward the work on a property-based bootstrap architecture of [12] by presenting a design and a prototype implementation of a Property-Based Boot Loader (PBBL) including efficiency measurements. We extend the concepts defined in the previous work to allow a realization based on a common PKI (Public Key Infrastructure). In addition, we solve the version rollback problem of [12] in such a way that it is secure even if the TPM owner of the client is untrusted. In contrast to the solution of [12], our proposal does not require an interactive protocol between the boot loader and a trusted third party (TTP) during the installation phase of the boot loader.

Moreover, we show how such a property-based bootstrap architecture can be integrated into typical centralized IT infrastructures of enterprises such that software updates can
2. RELATED WORK

There have been several proposals in the literature for protecting and proving the integrity of computing platforms based on cryptographic techniques and trusted components. Known aspects in this context are secure and authenticated (or trusted) booting: The former means that a system can measure its own integrity and terminates the boot process in case the integrity check fails, whereas the latter aims at proving the platform integrity to a (remote) verifier using binary attestation [3, 7, 18, 20, 26].

A more general and flexible extension to the binary attestation is property-based attestation (PBA) [17, 16, 12]: on a higher system level, attestation should only determine whether a platform configuration or an application has a desired property. PBA should allow the verifier to decide whether the target machine to be attested fulfills individual requirements with regard to a certain policy (e.g., it provides certain access control methods or complies with some privacy rules). This avoids revealing the concrete configuration of soft- and hardware components.

For most practical applications, the specific system or application binary configuration is less an issue for the verifier, and it is very efficient since the verifier has the burden of managing an enormous amount of configurations. Practically, properties change rarely compared to binaries on program updates.

A high-level protocol for property-based attestation is presented in [16]. The solution is based on property certificates that are used by a verification proxy to translate binary attestations into property attestations. In [17] the authors propose and discuss several protocols and mechanisms that differ in their trust models, efficiency, and the functionalities offered by the trusted components. In particular, [17] discusses how the Trusted Software Stack (TSS) can provide a property-based attestation protocol based on the existing trusted computing hardware without a need to change the underlying trust model. The basic idea is that, instead of sending the signed hash values of the binary code to the verifier, the trusted platform (i.e., the TSS) proves to the verifier in zero-knowledge that its current system state corresponds to the properties attested in the property certificates, which the platform sent with the proof to the verifier. In this context, [5] proposes a concrete cryptographic zero-knowledge protocol for anonymous property-based attestation.

Another refinement of property-based attestation is proposed in [12]. The basic concept of this approach is to enhance the boot loader such that it translates binary measurements into properties that have been certified by a trusted third party. Instead of binary measurements the properties (i.e., unique bit strings representing properties) are reported to the Platform Configuration Registers (PCR) in the TPM. Thus, the PCR values remain the same after soft- or hardware updates as long as a valid certificate for the new binary measurement exists that attests the same property to the updated component.

In [6] the authors reconsider a cryptographic protocol without involving a trusted third party. The idea of this approach is that the trusted platform proves to the verifier that its current configuration is within a set of platform configurations that all have the same properties without revealing the actual configuration of the platform. However, this requires modifications to the TPM interface.

The authors of [9] propose semantic remote attestation using language-based trusted virtual machines (VM) to remotely attest high-level program properties. The general idea is to use a trusted virtual machine (TrustedVM) that verifies the security policy of the machine running within the VM. Another work also focusing on different properties is the Usage Control (UCon) model based behavior attestation [2].

In [13, 14, 15] the authors propose a software architecture based on Linux providing attestation and binding. The architecture binds short-lifetime data (e.g., application data) to long-lifetime data (e.g., the Linux kernel) and allows access to that data only if the system is compatible to a security policy certified by a security administrator.

3. BACKGROUND ON PROPERTY-BASED BOOTSTRAP ARCHITECTURES

Our work is mainly based on [17, 12], hence we first recall the main aspects of property-based bootstrap architectures.

The general idea is to use a bootstrap architecture translating binary measurements (e.g., the hash values of binary codes) into properties. The translation is done before the operating system is bootstrapped and authenticated by the so-called Property Certificates issued by a TTP. Since the properties are extended into the PCRs, applications can use them through an unmodified TSS.

When using Trusted Computing functionality, some parts of the underlying IT infrastructure should fulfill certain requirements [12]. These requirements include availability of sealed data after a system update, scalability in the context of large enterprise environments, compatibility of existing operating systems and current available TCG-compliant products, security of property-based attestation and sealing, and freshness of properties (also denoted as the version rollback problem).

With the enhanced bootstrap architecture, certified software properties (represented by PCR values) do not change after an update. This is why software updates are transparent to existing operations such as sealing and attestation. However, this kind of transparency also raises the software version rollback problem, meaning the client is able to install an older version of a software but still claim having the same property on its platform. In order to deal with this problem, [12] proposes two solutions: while the first only works...
with remote attestation, the latter, which is based on TPM v1.2 [22, 23, 24] features, supports both remote attestation and sealing.

The main idea behind the first solution to version rollback is to extend the Certificate Revocation List (CRL) serial number into a PCR to allow remote entities to check the freshness of the CRL during attestation.

The second approach that supports sealing uses monotonic counters of the TPM, the sealing operation, and NV (Non Volatile)-Space to store the latest CRL serial number such that freshness can be checked. More concretely, [12] suggests to bind the state to a monotonic counter, to seal the resulting blob, and to store it into an NV-Space region of the TPM. To prevent TPM owners from deleting this state to reset the CRL version state, [12] suggests a protocol performed by the enhanced boot loader ensuring freshness of a newly installed CRL. However, [12] only describes the high-level approach. We take it a step further by realizing an efficient Property-Based Boot Loader (PBBL) and investigate the security impact on existing TPM functionality.

### 4. PROPERTY-BASED BOOT LOADER

In the following, we describe the Property-Based Boot Loader (PBBL) realization including the involved roles, the underlying trust model, and important design and implementation aspects in the context of centralized IT networks.

We adopt the two main requirements of [12]: Authentication of the software running on the client platform and preservation of privacy regarding the platform configuration.

#### 4.1 Involved Roles and Assumptions

In the context of property-based attestation and sealing, different parties with different interests in security and privacy have to communicate to each other leading into a more complicated trust relationship model than with binary attestation. The involved roles in this trust model are:

- **Identity-CA**: Similar to the TCG proposal, the Identity-CA (ICA) identified by the key pair \((PK_{ICA}, SK_{ICA})\) is responsible for providing identities or pseudonyms of platforms involved into the property-based attestation protocols. However, in a centralized enterprise environment where the IT department owns all client platforms, anonymity is not an issue. In contrast, an IT department normally is interested in a unique identification of platforms joining the company network. Therefore, in our context the role of the Privacy-CA\(^1\) is realized by the Identity-CA.

- **Property-CA**: The Property-CA (PCA) identified by the key pair \((PK_{PCA}, SK_{PCA})\) is a new role who issues property certificates based on a security policy defined by the enterprise. The local platform user needs to download the Privacy-CA an issue. In contrast to the solution proposed in [12], an online connection to the Property-CA is not required.

- **Verifier**: The verifier is the party who is interested in the configuration of the attested platform. It has to trust the TPM to correctly attest the PBBL’s binary configuration and the PBBL to correctly map binary values into properties. Moreover, the verifier has to trust the Identity-CA to only certify platforms owned by the enterprise and the verifier has to trust the Property-CA to only certify software updates according to the enterprise’s security policy.

#### 4.2 Extended Data Structures

We recall the important data structures suggested in [12] and highlight the modifications of our realization:

- **Property Certificate**: A property certificate issued by PCA vouches for a property of a software module. More concretely, it is used to bind a property to a hash value of a software binary. To allow identification of property certificates, the certificate serial number is also included. We denote a property certificate as:

\[
C = (PK_{PCA}, H(S), P, C.serno, \text{Sig}_{SK_{PCA}}(H(S), P, C.serno))
\]

where \((PK_{PCA}, SK_{PCA})\) is the key pair of PCA, \(H(S)\) is the hash value of the software binary, \(P\) represents the property, \(\text{Sig}_{SK_{PCA}}(x)\) is a signature on \(x\) using the signing key \(SK_{PCA}\) of PCA, and \(C.serno\) is a unique serial number of the certificate generated by PCA to identify each issued certificate.

- **Certificate List (CL)**: The CL is issued and signed by PCA. The CL has a unique serial number and holds a list of certificate serial numbers of certificates \(C\). Using the certificate list, both blacklist and whitelist models can be supported. In the context of blacklisting, the certificate list becomes a certificate revocation list (CRL) requiring a secure time source to enforce validity periods of property certificates. In the context of whitelisting, a secure time source is not required, since a whitelist does not normally increase over time\(^2\). The CL is denoted as:

\[
CL = (PK_{PCA}, CL.serno, List_{C.serno}, \text{Sig}_{SK_{PCA}}(CL.serno, List_{C.serno}))
\]

where \(CL.serno\) is the latest CL version number that \(PCA\) generated and \(List_{C.serno}\) the list of property certificates.

#### 4.3 Improved Version Rollback Prevention

As described in Section 3, the version rollback protection mechanism suggested in [12] was rather complicated, since it included monotonic counters, a sealing operation, NV-Space, and an interactive protocol ensuring the freshness of a newly installed CL. In our solution, we are using an improved design that only needs NV-Space, and no protocol

\(^1\)Privacy-CA is a trusted third party that certifies TPM genuineness and guarantees the pseudonymity of TPM identity.

\(^2\)Except the number of allowed configurations increases over time, of course.
is needed to ensure CL freshness during the PBBL installation. Therefore, we introduce a new PBBL version state blob \( B \), denoted as:

\[
B = [CL\_serno, rnd]_{PBBL}
\]

where \( CL\_serno \) is the latest CL serial number and \( rnd \) is a random value created by the PBBL during installation. The resulting blob is stored in the NV-Space of the TPM and bound to the binary configuration of the PBBL as indicated by the notation \([\ldots]\)\(_{PBBL}\).

The binding of the NV-Space region of \( B \) is realized by defining \( C_{PBBL} \) as the only configuration that overwrite \( B \). The TPM internally enforces this binding policy by access control mechanisms, therefore it is not required to explicitly seal the state blob. The advantages of using NV-Space instead of sealing are that with sealing, (i) everyone with access to the file system can perform a DoS, and (ii) sealing additionally requires the use of a monotonic TPM counter to prevent an attacker from replaying an old sealed blob.

However, the TPM does not prevent the TPM owner from completely deleting a NV-Space. This allows the owner to perform a replay attack by deleting the NV-Space of an existing PBBL and installing a new PBBL instance. This attack is prevented by the random value \( rnd \) that uniquely identifies a PBBL installation. When data is sealed also to this value, unsealing with another PBBL instance will fail, since it uses a different random value.

During bootstrapping, the PBBL reads \( rnd \) from NV-Space which is bound to \( C_{PBBL} \) (e.g., digest of PBBL binary and currently empty PCR\[t+1\] and PCR\[t+2\]), extends \( rnd \) to PCR\[t+1\] to prevent unsealing within a new PBBL instance. This attack is prevented by the random value \( rnd \) that uniquely identifies a PBBL installation. When data is sealed also to this value, unsealing with another PBBL instance will fail, since it uses a different random value.

4.4 Extended IT Infrastructure

Figure 1 illustrates the extended IT infrastructure required to use the property-based bootstrap architecture within an existing environment.

![Figure 1: Extended IT infrastructure with additional components needed for property-based bootstrap](image)

Additional components, compared to the conventional IT infrastructure, are the Property-CA and the PBBL. The Property-CA is responsible for generating property certificates of software modules to be certified. Additionally, all TPM-enabled client platforms are equipped with the PBBL that is responsible for translating binary values into properties and to ensure the freshness of the CL.

For simplicity, we assume for our envisaged use cases that the properties considered by the IT department represent Common Criteria Evaluation Assurance Levels (EAL) of the underlying operating system. That is, an operating system that has been evaluated according to EAL4 is represented by the property 0x04.\(^3\) After successfully installing and configuring the Property-CA and PBBL, a software update is handled as follows:

- **Property Certificate Issuing:** If the IT department decides to provide a software update, a new property certificate is generated. The software update is now distributed together with the corresponding property certificate.
- **Software Update:** If the operating system on the client platform is updated, it only has to be ensured that the corresponding property certificate is available to the PBBL.
- **Bootstrapping:** During bootstrapping, the PBBL measures the software to be loaded using a conventional hash function. It then checks whether a valid property certificate for the software exists. If it exists, it writes the certified properties and the identity of the Property-CA to the PCRs.
- **Configuration Revocation:** If a software version should not be used any more, e.g., due to a known exploit, the Property-CA revokes the corresponding property certificate using the CL. Then the PBBL should ensure that the revoked software cannot be used to unseal data anymore. Moreover, if the PBBL detects a replay attack against the CL, it also invalidates the PCRs such that unsealing of data becomes impossible.

4.5 PBBL Design Overview

In general, the property-based bootstrap architecture is separated into two packages:

- The first package realizes the Property-CA functionality such as setup, property certificate creation and revocation, and the creation of identities. The implementation aspects of this package are described in Section 4.6 in more details.
- The second package is the PBBL itself that is executed on the client side. The most important use cases to be realized by the PBBL are initialization, bootstrapping, property-based attestation, and property-based sealing. However, the last two use cases are compatible with the corresponding binary versions. Therefore, we only describe in Section 4.5.3 and Section 4.5.4 the additional steps required to interpret the semantics of the used PCRs.

Assuming that PCR\[t\] is the first PCR index used in property mode and PCR\[0..t-1\] are used by the components preceding the PBBL, the following PCRs are used:

- **PCR\[t\]**: Stores the public key of the Property-CA. It allows remote entities to identify the Property-CA that issued property certificates for the attesting platform.
- **PCR\[t+1\]**: Stores the random value \( rnd \) (used to prevent a reset attack by clearing the PBBL’s NV-Space.).

\(^3\)Note that the concrete encoding of properties is not an issue here. It only has to be defined by the IT department.
4.5.1 PBBL Initialization

When the PBBL is executed for the first time, it initializes the version state block B used for CL rollback prevention. As depicted in Figure 2, the PBBL first gets the available NV-Space by using the TPM command TPM_GetCapability and creates a new NV-Space that is bound to the platform configuration C_{PBBL} using TPM_NV_DefineSpace. Then creates a random number \( \text{rnd} \) identifying the current installation using TPM_GetRandom, and checks the availability of \( PK_{PCA} \) and the CL, verifies the CL. Finally, writes the state blob \( B \) including the CL serial number \( \text{CL.serno} \) and the installation identifier \( \text{rnd} \) into the NV-Space using TPM_NV_WriteValue.

Since any request to read and write the NV-Space created by the PBBL will be checked against \( C_{PBBL} \) by the TPM, the access of \( B \) is restricted only to the PBBL. As already pointed out in Section 4.3, deletion of the NV-Space region of \( B \) by the TPM owner cannot be performed by a replay attack. Since the new PBBL installation will use another random value, unsealing will fail.

Note that overwriting the NV-Space region of \( B \) from within a loaded operating system won’t work, since the PBBL extends PCR[\( t+1 \)] and PCR[\( t+2 \)] before booting the operating system and thus the TPM will deny write access to that NV-Space.

4.5.2 Bootstrapping

In bootstrapping mode, the PBBL first reads the Property-CA’s public key \( PK_{PCA} \) from the Property-CA’s identity certificate and uses it to check integrity of the CL. Then the PBBL reads the version state blob \( B \) from the TPM NV-Space using the TPM command TPM_NV_ReadValue to check whether the CL.serno matches. If \( B \)’s serial number is smaller (CL update), the PBBL will update the CL.serno in the state blob \( B \) (updateStateBlob()) by writing the new \( B \) into the NV-Space using TPM_NV_WriteValue. The next step is to extend \( PK_{PCA}, \text{rnd} \) and \( \text{CL.serno} \) into PCR[\( t \)]. PCR[\( t+1 \)] and PCR[\( t+2 \)] respectively using TPM_Extend. If \( B \)’s serial number is larger (CL rollback), the PBBL invalidates PCR[\( t \)] to PCR[\( t+2 \)] (invalidatePCR()) by extending it with zero and continues to bootstrap the selected operating system. In a last step, the PBBL will measure and verify the target software to be booted against property certificates \( C \). If any mismatch occurs, the PBBL asks the user to update the CL, property certificates, or the software. Otherwise, the corresponding property is extended into the following PCR using TPM_Extend (refer to Figure 3).

In this way, the PBBL can make sure that the user can authenticate the booted system successfully by the above PCR values.

4.5.3 Property-Based Attestation

As depicted in Figure 4, the property-based attestation protocol is compatible with the binary-based protocol. Here, the key pair \( (PK_{TPM}, SK_{TPM}) \) generated by the TPM is still needed for TPM identity and data authentication. However, in order to achieve both the availability and security requirements, the verifier has to correctly select and interpret the attested PCR values. The PCR values of interest in the context of remote attestation are PCR[0..\( t-1 \)] including the binary values of the PBBL, PCR[\( t \)] containing the \( PK_{PCA} \), PCR[\( t+2 \)] containing the \( CL.serno \) and the attested properties \( P \) of the loaded software. PCR[\( t+1 \)] is not important here, since it only includes the installation identifier \( \text{rnd} \) needed for property-based sealing.

4.5.4 Property-Based Sealing

Another aspect of our approach is that it is transparent to critical TCG functions such as sealing and binding. As PCR[\( t+3 \)] and above will still hold the same values after a software update, representing the software’s properties, these functions won’t be affected. However, to prevent attackers by bypassing security policies by changing the PBBL or by deleting the NV-Space to rollback software versions, software should also be sealed to all other PCRs except PCR[\( t+2 \)] which includes the current CL serial number.

4.6 Implementation

The PBBL has been implemented on an x86 architecture system featuring a TPM of version 1.2 and is described in the following section. According to the PBBL design, its implementation has been divided into two parts, the Property-CA of the IT-department and the boot loader itself.

4.6.1 Property-CA

The complete functionality of the Property-CA has been integrated into one tool used to create and maintain the Property-CA. The tool mainly serves as an interface to the OpenSSL tool set which is the core component used to manage the PKI.

Our CA setup includes a Root-CA and a subordinate Property-CA (certified by the Root-CA). Upon Root-CA
creation, a self-signed certificate is issued. In the second step, the Property-CA creates a certification request, which is signed by the Root-CA for authorizing the Property-CA to sign certificate requests on its own. If the CAs are properly set up, the Property-CA can issue new property certificates.

We use X509 version 3 (X509v3) certificates to implement the property certificates. As public key certificates are used to certify keys, not properties, we use a dummy public key to be certified by the Property-CA in every property certificate. To add the properties to each certificate, we define three new X509v3 extensions:

- "softwareInfo" is a textual description of the software to be certified
- "softwareProperty" includes the binary representation of the software’s property
- "softwareDigest" includes the digest of the software’s binary

These extensions are added to the X509v3 certificate upon the certification request of a property certificate and bound together with the dummy key mentioned previously. The certificate request is signed by the Property-CA, creating a valid property certificate representing the binary-to-property binding.

For compatibility reasons, we reuse the existing certificate revocation mechanism provided by the OpenSSL library using a black list model for the Certificate List, more precisely a Certificate Revocation List (CRL).

4.6.2 Boot Loader

The bootstrap architecture specified by the TCG includes essentially 3 components. The Core Root of Trust for Measurement (CRTM), the boot loader, and the operating system to be loaded. Here, each component measures the next by calculating the digest of the executable, extends it into a particular PCR and finally passes control to the next booting component. This way, a chain of trust is built throughout the whole bootstrap architecture.

To simplify the PBBL realization, we build it on top of a Linux kernel instead of writing/enhancing an existing TPM-enabled boot loader. The main reason for this decision was not to integrate many crypto, certificate management and TPM functions needed into the boot loader. Therefore, we insert an additional stage into the chain of trust (see Figure 5). It includes an intermediate Linux kernel executing our PBBL that is implemented as a user-space process.

In the property-based bootstrap architecture, the CRTM as well as the TrustedGRUB [19] perform binary measurements. The CRTM measures TrustedGRUB, then TrustedGRUB performs binary measurements of the intermediate Linux kernel binaries and the PBBL binary and configuration data. All of these binary measurements are extended into the appropriate PCR[0, · · · , t-1]. Loaded by TrustedGRUB, the intermediate Linux kernel executes the PBBL, which starts the property-based boot process.

In the first step, the PBBL makes sure that the most recent CRL is installed by comparing it to the state blob inside TPM’s NV-Space. Depending on the result of this operation, it either updates the state blob and continues in case a new CRL has been found on disk, or simply continues if the CRL has not been updated. In the second step...
it measures each binary of the target Linux kernel (and optionally arbitrary files) and looks for property certificates, the value of “softwareDigest” X509v3 certificate extension of which matches the binary digest. In the third step, the PBBL verifies each property certificate. Verification is a two-step process. First, the Property-CA’s signature on the certificate is verified. Second, the serial number of the certificate is checked against the CRL to see if the certificate has been revoked previously. If any of the above steps fails, the boot loader invalidates the PCR. In the last step, the PBBL invokes the kexec kernel-mechanism. This mechanism loads another kernel, in our case the target kernel, into memory and executes it. The target Linux operating system is now booted and ready for use. Note that the target operating system can be any kernel, e.g., Windows or a secure hypervisor.

The PBBL itself has been written in C++ and depends on several software components. The TinyXML library integrated into the boot loader is used for parsing and modifying the boot loader configuration file. All the TPM functionality is provided by an object-oriented TCG Software Stack written in C++ developed by [1]. The third required library is the OpenSSL library. It provides all the functionality to verify signatures and deal with X509v3 certificates. Last but not least, the PBBL implementation also depends on the kexec kernel-mechanism to load the target kernel, which was introduced to the Linux kernel in version 2.6.

4.6.3 Detailed View

We now describe the decomposition of the PBBL into its main components as illustrated in Figure 6:

Main. The main application loop. It controls the interactions among the other components.

Translator. The Translator’s main task is to translate binary values into properties. It is also responsible for checking the freshness of revocation data, creating and updating the state blob $B$.

Selector. The Selector’s main task is handling the boot loader’s configuration file. It is responsible for loading and storing data from and to the configuration file and provides methods that read configuration values. Moreover, the Selector presents the boot selection screen to the user and returns the selection back to the main application loop.

PropertyCert. This component simply represents a Property Certificate. It publishes methods to receive X509v3 extension values and returns the verification result using the OpenSSL library.

SSL. This component encapsulates functionality provided by the OpenSSL library. It includes complex functions such as extracting a public key from a certificate as well as the CRL verification.

4.6.4 Performance Measurements

We evaluated the PBBL to see how it performs in comparison to a normal boot involving only TrustedGRUB and the target Linux OS. Table 1 shows the results of a performance test on the modified chain of trust introduced in Figure 5 and a minimal setup of only the target kernel and the initial ramdisk image to be processed by the PBBL.

<table>
<thead>
<tr>
<th>#</th>
<th>Normal Bootstrap</th>
<th>Property-Based*</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>49.5s</td>
<td>73.3s (3.5s)</td>
<td>23.8s</td>
</tr>
<tr>
<td>2</td>
<td>46.0s</td>
<td>73.3s (3.6s)</td>
<td>27.3s</td>
</tr>
<tr>
<td>3</td>
<td>46.0s</td>
<td>77.2s (2.9s)</td>
<td>31.2s</td>
</tr>
<tr>
<td>$\forall$</td>
<td>47.2s</td>
<td>74.6s (3.3s)</td>
<td>27.4s</td>
</tr>
</tbody>
</table>

Table 1: Performance Comparison of a normal boot and a property-based boot using an intermediate kernel

The results show that the PBBL costs about 3-3.5 seconds, only an average 12% of the additional boot time. However, the whole property-based boot process takes about 25-30 seconds longer than a normal boot process. The performance of the PBBL itself is actually not very critical, since the most time-consuming part of the enhanced architecture is intermediate kernel boot process. Therefore, the performance overhead can be expected to be reduced to the SHA-1 measurements and certificates verifications when integrating the PBBL functionality into a common boot loader.

Further performance tests showed that our solution also scales. Figure 7 depicts these results, showing an almost linear increase on file processing, including SHA-1 measurement of binary, lookup of a compatible property certificate and verification of both the property certificate and the corresponding CRL. The files were taken from the /bin directory of a common Linux distribution, and are rather small in file size (about 1K-200K).

These results lead to the conclusion that the PBBL should be integrated into an enhanced boot loader such as TrustedGRUB. Another solution would be to replace TrustedGRUB’s stage two by the PBBL. This step should be easy, as the PBBL is already a single module. Either way, the intermediate kernel could be removed and thus the boot time can be dramatically reduced.

4.7 Evaluation and Security Analysis

In this section we informally analyze the security aspects of our property-based boot architecture (including the requirements given in [12]).

Availability. Our property-based approach uses properties instead of binary hash values. The PBBL extends the
software’s properties into the appropriate PCR values instead of their binary hash values. Performing a software update does not change the PCR values as long as an appropriate property certificate exists. Therefore, sealed data is still available even after a software update.

**Scalability.** The main motivation behind our property-based attestation and sealing approach is to delegate this decision from the verifier respectively client platform boot loader to another trusted entity. On the one hand, our approach removes the complexity of managing all combinations of binary hash values from the verifying entity. Thus verifying clients can focus on the properties they are interested in. On the other hand, our approach eases software updates, because rescaling of the data is not required any more.

**Compatibility.** The PBBL makes use of existing hardware and software, as it is implemented as a user-space application for the Linux operating system on a common x86 desktop computer system. The applications and TSS that use TPM functionality such as sealing and attestation can benefit from the property-based bootstrap architecture without modifying the interfaces. However, minimal extensions are needed in the TCG attestation protocol, because the verifier has to check the freshness of the attested CRL serial number. Despite these software modifications, the PBBL still fulfills the requirement of compatibility.

**Freshness.** To prevent a version rollback of the loaded software, the PBBL uses a certification list (blacklist or whitelist) to distinguish between valid and invalid software versions. That is, the PBBL only attests properties of software versions that are marked as valid. To prevent a rollback of the CL, more efforts are required, since the PBBL cannot distinguish the differences between the CL upon a fresh installation and an overwritten old version. A malicious TPM owner may delete the state B from NV-Space, making the PBBL believe that it has currently been installed and must go through initialization, storing the CL.serno from the CL on disk in NV-Space again and thus leaving the system vulnerable to an improved version rollback attack. Therefore, we store the CL serial number CL.serno together with a random number rnd created during PBBL initialization using an NV-Space that is only writable by the PBBL itself. The CL serial number CL.serno is used to prevent a replay attack on the CL where the CL on disk is simply replaced by an older one, while the random number rnd is used to bind sealed data to a particular PBBL installation. During the boot process, the PBBL compares the CL.serno to the serial number of the CL found on disk and extends it into PCR(t+2), preventing successful CL rollback attacks.

Furthermore, it reads and extends rnd into PCR(t+1), preventing successful unsealing operations after a deletion of the NV-Space by the TPM owner and the automatic recreation of B by the PBBL.

**Security of Attestation and Sealing.** It’s important that the security properties of the existing critical TCG functionality, such as attestation and sealing, must not be negatively impacted. As our approach does not modify the TCG sealing operation and the TCG attestation protocol, the user only needs to choose the right PCRs to bind sensitive data, these modifications don’t have an impact on the security properties of these functionality. However, the security of our architecture depends on the correct implementation of the PBBL itself and a correctly functioning PKI.

**Privacy Issues.** The client’s privacy is protected in two ways by our approach. As there is no interaction with the Property-CA, the client’s identity is never revealed to the Property-CA. The client’s platform configuration is also kept private during attestation, as only the property names of the software are sent to the verifier, instead of the binary hash values of the software which could be used to identify the software running on the client’s platform. As there is infeasible to conclude from property to binary hash value on the verifier’s side, no information on client platform configuration is disclosed.

5. PROPERTY-BASED SECURE BOOT

A trusted boot process as proposed by the TCG based on Trusted Platform Module (TPM) allows remote parties to verify a platform configuration [13, 12]. But it does not provide a secure boot process, i.e., allowing (without additional trusted hardware) users to verify the trustworthiness of their local computing platforms. Various solutions have been proposed in this area. In [26], a cryptographic coprocessor is first proposed to verify every component to be used before loaded by checking the software’s signatures against known values. Later, AEGIS [3, 4] proposed a modified secure bootstrap process, providing local integrity verification and remote recovery by using the BIOS and additional ROM. The transitive trust is also provided here by using the component booted first to verify the integrity of the component executed next.

Mobile Trusted Module (MTM) [25, 21] is the TCG solution for mobile devices (e.g., mobile phones). In general, MTM provides a similar set of functions as conventional TPM. However, the MTM is rather a functionality than a hardware module, providing more flexible implementation options for its manufacturers. Two types of MTMs are defined in the TCG specification: while the MLTM (Mobile Local-Owner Trusted Module) is similar to a TPM, the MRMT (Mobile Remote-Owner trusted Module) provides in addition some functions to realize a secure boot.

The MTM secure boot approach improves secure boot proposed by the AEGIS in two ways. First, the MTM approach achieves initializing code immunity by using RTV (Root of Trust for Verification) and RTM (Root of Trust for Measurement) with minimum functionality (only measurement and verification of itself and the next booting module) instead of a modified BIOS to avoid system compromise caused by wrong BIOS update. Second, the MTM provides a way to transform external certificates including verification keys from different parties into internal certificates inside MTM, which only requires one embedded public key inside the

![Figure 7: Performance of PBBL with increasing number of files](image-url)
the MTM for verification. This guarantees the secure storage of verification key and releases the platform’s burden of keys and certificates management.

However, the secure boot process of MTMs is still based on binary hash values with all its side effects. Therefore, a property-based secure bootstrap process would be an appropriate solution providing end-users both, security and availability of their data.

Another difference between an MRTM and a TPM is that identity privacy of MRTMs is not necessary [8]. As our work also mainly focuses on platform configuration privacy and can mitigate the side effects of binary based measurements mentioned in Section 1, they’re naturally supplemented.

5.1 Secure Boot with MRTM

MRTMs implement secure boot based on the so-called RIM (Reference Integrity Metric) certificates including an entry defining the prerequisite PCR state and another entry referring to a new PCR value. Moreover, MRTMs include at least one public key (verification key) used to verify RIM certificates.

If a MTM is in secure boot mode, extending PCRs is only possible using the VerifyAndExtend function. Using a RIM certificate as an input value, the MTM verifies the RIM certificate, checks whether its current PCR state matches the prerequisite PCR state of the RIM, and, if both was successful, extends the new PCR value(s) defined in the RIM certificate. This way, a PCR value can only be extended into a PCR, if a valid chain of RIM certificates exists. Note, however, that the measurement of the module to be loaded has to be done by the boot loader before the corresponding RIM certificate is loaded into the MTM.

This approach is more flexible than simple binary reference, because it allows to define a prerequisite PCR state and allows updates of a component without updating the RIM certificates of the following components in the chain of trust.

5.2 Realization

Based on the secure boot concepts described above, our concept of property-based attestation and sealing by mapping binary values into properties can easily be applied to MTMs. In fact, both approaches are already similar, because (i) the secure boot concept is already based on certificates and (ii) a trusted third party to create the RIM certificates is already involved.

To realize property-based secure boot based on the VerifyAndExtend function of MTMs, the issuer of the RIM certificates only has to replace the binary values of the RIM certificate, i.e., the prerequisite PCR state and the new PCR value, by abstract properties. After a software update, the issuer only has to create a new RIM certificate that maps the new binary values into property values to be extended. Since the mapping from binary hash values into property-based RIM certificates has to be done by the boot loader, no changes of the existing RIM structures and MTM commands are required. Furthermore, with existing MTM function InstallRIM, the integrity of the new RIM certificate can be protected by the verification key in the MTM.

6. CONCLUSION AND FUTURE WORK

In this paper, we extended and improved the previous work on property-based bootstrap architectures [12] with respect to different aspects. In particular, we strongly improve the solution for software version rollback problem. Moreover, we present a proof of concept implementation of our Property-based Boot Loader. Our solution for a property-based boot loader can be easily integrated into existing IT infrastructures allowing enterprises to deploy Trusted Computing technology without facing the typical problems of binary attestation. More concretely, software updates can efficiently be distributed in this environment without violating security or availability requirements on (security critical) data and services. Our implementation results show that the inclusion of the property-based approach into a boot loader can be efficiently used in practice.

Finally, we showed how our concept can be applied to the secure boot mechanisms as defined in the Mobile Trusted Module (MTM) specification to realize a "Property-based Secure Boot". This is particularly important, since with secure boot, end-users can rely on the fact that a loaded system is also in a trustworthy state.

The integration of PBBL into a real boot loader is one of our main future goals. It can strongly improve booting speed, but more importantly, it can be used to build a smaller Trusted Computing Base (TCB) without involving an additional intermediate Linux kernel.

Also as mentioned earlier in Section 4.3, the PBBL still relies on binary measurements of previously booted hard- and software components, which does not scale well. One solution is to migrate the property-based bootstrap architecture to Dynamic Root of Trust for Measurement (DRTM) [11, 10]. When the DRTM call is triggered, the CPU measures the DRTM, which then measures the property of the next booting component (e.g., Virtual Machine Monitor) or security-sensitive codes. In this way, the update of other previously booted components will not affect the scalability of the property-based bootstrap.

Another research challenge concerns the meaning of properties. In our approach, we define the property in a simple scheme based on the evaluation (more concretely EAL) of the operating system to be loaded and actually delegate the interpretation of property to a trusted third party (the issuer of Common Criteria security evaluation), because the meaningful definition of properties is out of scope of this work. Last but not least property-based policy definition and enforcement is a further interesting research issue.

7. REFERENCES


