SCOBA: Source Code Based Attestation on Custom Software [*]

Liang Gu[‡], Yao Guo[‡][†], Anbang Ruan[§], Qingni Shen[§],Hong Mei[‡]
[‡]Key Laboratory of High Confidence Software Technologies (Ministry of Education),
[‡]Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing, China
[§]School of Software and Microelectronics, Peking University, Beijing, China
[‡]{guliang05,yaoguo,meih}@sei.pku.edu.cn,[§]{ruanab, shenqn}@infosec.pku.edu.cn

ABSTRACT

Most existing attestation schemes deal with binaries and typically require an exhaustive list of known-good measurements beforehand in order to perform verification. However, many programs nowadays are custom-built: the end user is allowed to tailor, compile and build the source code into various versions, or even build everything from scratch. As a result, it is very difficult, if not impossible, for existing schemes to attest the custom-built software with theoretically unlimited number of valid binaries available. This paper introduce SCOBA, a new Source COde Based Attestation framework, to specifically deal with the attestation on custom software. Instead of trying to obtain a know-good measurement list, SCOBA focuses on the source code and provides a trusted building process to attest the resulting binaries based on the source files and building configuration. SCOBA introduces a *trusted verifier* to certify the binary code of custom-build program according to its source code and building configuration. For custom-built software based on open-source distributions, we implemented a fully automatic trusted building system prototype for SCOBA based on GCC and TPM. As a case study, we also applied SCOBA to Gentoo and its Portage, which is a source code based package management system. Experimental results show that remote attestation, one of the key TCG features, can be made practically available to the free software community.

[*]This work is supported by the National Basic Research Program of China (973) under Grant No. 2009CB320703, the Science Fund for Creative Research Groups of China under Grant No. 60821003, National Key S & T Special Projects under Grant No. 2009ZX01039-001-001 and the National High-Tech Research and Development Plan of China under Grant No. 2007AA010304 and No.2009AA01Z139, and National Natural Science Foundation of China under Grant No. 60873238 and No. 60903178.
[†]corresponding author

Categories and Subject Descriptors

D.4.6 [**Operating System**]: Security and Protection—*Authentication, Invasive software*; D.2.4 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Security

Keywords

Remote attestation, custom software, trusted computing, free software, property-based attestation

1. INTRODUCTION

Many IT systems nowadays are conducted on open computer platforms across heterogeneous domains or over the public Internet. Entities in such an open, distributed and dynamic environment usually behave on their own behalf and may not trust each other for mission-critical operations or transactions. Remote attestation provides an important way to establish trust on parties in an open network. In Trusted Computing Group's trusted computing standard [23], remote attestation allows a challenging platform, usually referred to as a *challenger*, to verify the configuration integrity of a remote platform (i.e., an *attester*). Recent years have witnessed various evolutions out of the basic TCG attestation in many dimensions, including IMA [21], program semantics attestation [12], security policy enforcement [14], property attestation [4, 19], BIND [22], remote attestation on program execution [11], and so on.

Most of the existing remote attestation schemes are based on the integrity measurement of programs and configurations. The size of known-good measurements greatly limits the practicability of the existing attestation schemes. For example, free software and open-source software make it difficult, if not impossible, for existing attestation schemes to verify the genuineness of the corresponding binary code.

From the perspective of software deployment, there are usually two types of software: custom software and prepackaged software. Many software nowadays are based on open-source software distributions, which greatly accelerate the software development process. However, since the users are in control of all the source files, they are able to tailor, configure and build their own executables to be deployed in their own environment. Even worse, they could also modify the source files at their own discretion, which would make the situation worse for attestation schemes. Custom software can be configured and tailored according to the end user's requirements which cannot be predicted by the software provider. For example, Linux kernel can be configured and built for each and different platform, with different requirements specified by the users. The result is that even all users download a specific software from the same trusted source website, the executables they built themselves could all differ from each others.

Existing remote attestation schemes are not adequate to verify these custom-built software, mainly because it is impossible to hold a known-good measurement database for so many different programs of unpredictable versions. Although existing property-based attestation schemes [4, 19] introduced the concept of attesting programs based on their properties, these stated properties are still tied to the binary code. As a result, these property-based attestation scheme still need a giant known-good binary database, which is still not able to handle the custom software.

To deal with these challenges from custom software, we propose SCOBA, a new Source COde Based Attestation framework to solve the above problems and initiate an effort for applying remote attestation, one of the key TCG features, in the free software community. The rationale of our scheme is to link specified binary code of the customized software with its source code, and certify the generated binary code of the software according to both its source code and building configuration.

In order to validate the generated binary code of custom software, we introduce a Trusted Building System (TBS) to enable a trusted building process for compiling the custom software (Figure 1). In the trusted building process, the source code of the target program is tailored according to the end user's requirements and it is complied into binary code with the TBS, in which the binary code is bound with its corresponding source code and building configurations. The building process can be attested to prove the validation of the generated binary code. With the generated binary code and its corresponding source code as well as building configuration, a trusted verifier is introduced to certify the property of the custom software (step a and step b in Figure 1). At runtime, challenger may use the certificate to enable remote attestation on the custom software (step c and step d in Figure 1). So the trust chain of our attestation scheme can be built from the TPM to the building process, and finally to the attested custom software at runtime.

This paper makes the following key contributions:

- SCOBA solves the problem of known-good measurements database for custom software. With the proposed framework, it becomes practical to attest customized software in open networks. To our best knowledge, it is the first effort for employing attestation to enhance trust establishment on customized software, especially for customized open-source software.

- The proposed Trusted Building System enables another party to validate and certify the generated binary code of custom software according to its source code and building configuration. Existing solutions can not attest custom software, because it has no way of validating the binary code of custom-built software.

- The source code based approach is a more practical way to obtain the software property. As SCOBA binds the source code files and building configuration of the customized software, the trusted verifier may obtain its property by evaluating and testing these information.

- The trusted verifier in SCOBA serves as the verification agent, which can be customized to accommodate different types of software. As a result, SCOBA provides a flexible framework, which can be customized according to different types of software development process, as demonstrated by the case study on Gentoo.

The paper is organized as follows: we will give a brief introduction on background in Section 2. Section 3 introduces the design of SCOBA. Section 4 presents the implementation and evaluation on the prototype of SCOBA. Section 5 introduces the application of SCOBA in Gentoo. Section 6 introduces the related work. Section 7 discusses the possible solutions for improving our scheme and its application. Section 8 concludes the paper and discusses the future work.

2. BACKGROUND

2.1 Custom Software

From the perspective of software deployment, there are usually two types of software: custom software and pre-packaged software. Custom software, which is also called bespoke software, allows end user to design and implement software based on its own requirements. Pre-packaged software, or "off-the-shelf" software is released by software provider with specified configurations, such as the installation packages under Windows, rpm packages and Debian packages under Linux.

Sometimes custom software is referred to as configured software, or customized software, which is tailored or customized from the original version of delivered software. The custom software starts from an existing structure and it is flexible for various requirements. Free software and open software are the most widely available custom software. Like the Linux kernel, end users may modify and configure the free software at will to satisfy their specific requirements. However, such flexibility results large number of unpredictable versions for binary code of free software.

In this paper, we will consider two kinds of custom software: custom-built software–customizing without modifying the source code files; fully custom software–customizing with modifications on source code files. For the first type, the users customize the software distribution before building, but do not modify individual source code files that are attained from trusted parties. For the fully custom software, users can modify the source code of the original software. For the custom-built software, as it is supposed to have fixed source code, SCOBA is able to automatically certify this type of customized software. For the fully custom software, SCOBA may have to employ experts or more sophisticated certification techniques to certify these modified source code, such as model checking and testing.

When considering their tailoring platform, compilation platform and execution platform, the custom software deployment can either take place all on the same platform, or it could be performed on separated platforms. For example, the compilation and execution are carried out on separated platforms; the source code is tailored and built on a separated platform according to the end user's requirements.

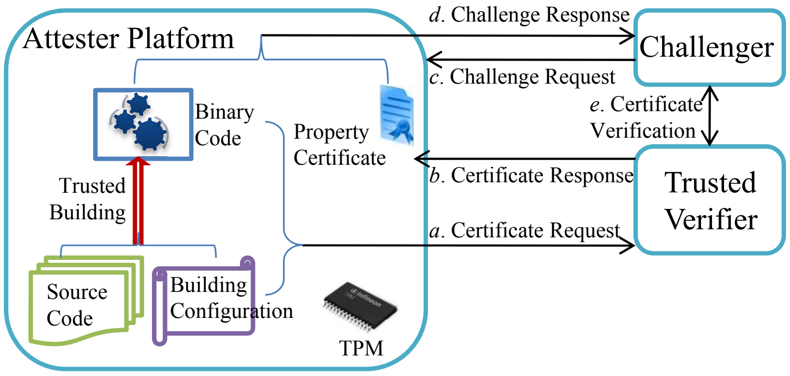

Figure 1: The overview of SCOBA

These approaches may require different designs and implementations of secure execution environment for the trusted building process in our scheme, and this will be discussed in Section 3.3 and Section 4. For most existing customized software, the first approach with the same platform is the most frequently used, thus we will focus on it most of the time and point out the difference if separated platforms might be used.

2.2 Dynamic Root of Trust

The TCG specifications introduce authenticated boot (or secure boot) to prove or guarantee that the system is booted into a secure state. The authenticated boot and secure boot provide a static root of trust based on the TPM. However, the static root of trust can not guarantee the security of a runtime system. With AMD's Secure Virtual Machine (SVM)[2] and Intel's Trusted Execution Technology (TXT)[13], it becomes practical to provide a dynamic root of trust for runtime system. The dynamic root of trust can strongly support a secure domain for dedicated system at runtime. Some studies for leveraging dynamic root of trust to provide secure execution environment have already been proposed, such as OSLO [15], Flicker [17] and TrustVisor [16]. For custom software delivered on the same platform, the end user may run the compilation process in the secure domain, which is supported by a dynamic root of trust.

3. SCOBA DESIGN

3.1 Attestation Framework

To provide remote attestation for custom software, we propose a new source code based attestation framework called SCOBA, which is illustrated in Figure 2. Instead of trying to obtain a list of known-good measurement list, SCOBA focuses on the source code, and provides a trusted building process to verify the resulting binaries according to the source files and building configuration.

Three parties are involved in SCOBA: the challenger, the attester and the trusted verifier. A typical scenario is as follows: the builder configures, tailors and builds a custom software P according to the challenger's requirement; the trusted verifier certificates the custom software by checking its source code, compiling configuration, binary code and records of building process; the builder delivers the custom software to the challenger with its certification; at runtime, the challenger wants to attest the property of this customized program; the challenger and attester will carry out the attestation procedure for this free software with the help of the trusted verifier.

Attester

The attester is the end user of a customized program P, which is executed on the attester platform. The attester customizes the source code of program P and takes a trusted building process to compile the source code into binary code. The trusted building process is introduced in Section 3.3. The attester employs a trusted verifier for property certification on the tailored program. The attester platform is supposed to be equipped with TPM.

Challenger

The challenger needs to attest the customized software being executed on the attester platform. The challenger requests the attester platform to return the integrity measurement and certificate of the target program. With these results, the challenger requests the trusted verifier to verify the certificate to determine the property of the target program.

Trusted verifier

The trusted verifier carries out two key tasks: property certification on customized software; runtime certificate verification. When the attester finishes the trusted building process, it requests a property certification on the customized program by sending all required source code, binary code and other records of trusted building process to the trusted verifier. The trusted verifier checks all these files and records to conclude with certain property for the customized program. At runtime, the challenger requests the trusted verifier to verify the certificate of the target program with specific integrity measurement. The trusted verifier can be a Trusted Third Party that issues property certificates and verifies certificates. The original provider of the program from which the customized software originates, can naturally serve as the role of trusted verifier.

3.2 Attester Platform

In the SCOBA framework, we assume that the attester platform (shown in Figure 2) is equipped with TPM, TXT[13] or SVM[2], the Secure Virtual Machine, the TCG software stack and an Attestation Agent, as well as a trusted building

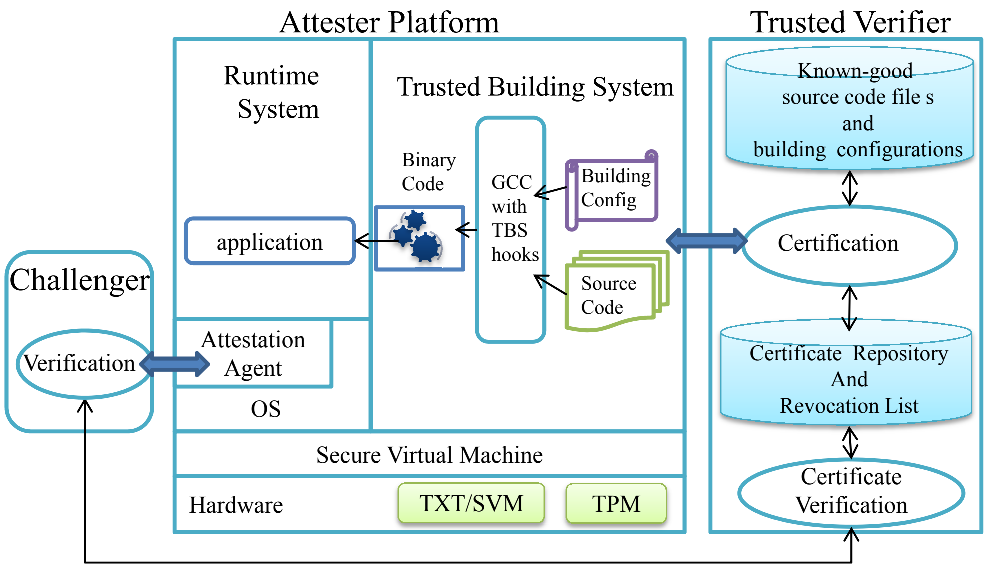

Figure 2: SCOBA Framework

system. The attester has privileges in controlling the software system on its platform. The attester platform may provide both static root of trust and dynamic root of trust. The challenger can establish trust on the integrity of a trusted domain based on Secure Virtual Machine with a dynamic root of trust.

Two separated domains are supported by the Secure Virtual Machine on the attester platform. One domain is a normal domain for ordinary operating systems. We introduce an attestation agent as the kernel module in the OS for runtime monitoring and recording target applications. The attestation agent is also responsible for communications between the challenger and attester platform. When the attestation agent receives the attestation request from the challenger, it records the states of target program and returns the target program's measurement and property certificate to the challenger. With the support of TPM and Secure Virtual Machine, the integrity of the attestation agent can be recorded for attestation each time before it starts.

The other domain is a secure domain that hosts a Trusted Building System (TBS). The TBS provides a trustworthy process for building these customized source code into binary code. The Secure Virtual Machine leverages the TXT/SVM facilities to provide a trusted domain for TBS. The building process can be attested to prove its trustworthiness.

The attester platform in Figure 2 is designed for customized software deployed on the same platform (Section 2.1). The TBS is supported by a dynamic root of trust. If it is carried out on separated platforms, the attester platform can have only the normal domain with attestation agent, while the TBS can be host on another separated platform. So TBS can run in a separated environment and its trust chain can be built on a static root of trust.

3.3 Trusted Building System

The Trusted Building System provides a trusted compilation process. A compilation process is considered trusted if the integrity of its execution can be attested to be without tampering. As a result, the compiled binary can be guaranteed to be generated from the input source code with specified configuration. In our scheme, the execution of TBS is protected by the secure domain, which can be set up at runtime based on a dynamic root of trust. TBS is supposed to be the minimal size for carrying out a compiling task and it is practical to implement the TBS with a thin OS and necessary compilation tools, e.g., the Linux From Scratch [1].

TBS records the states of all required proofs for program property certification. At the beginning of the trusted compiling process, TBS needs to record the building configuration. TBS records the state of compiled source code and output binary code files in a fine-grained and exact way according to their compiling order: the state of each source code file is recorded immediately before compilation; the state of each binary code file is recorded immediately when it is output by the compiler; meanwhile, TBS also binds the binary code file's measurement with the records of its corresponding source code files. In order to guarantee the integrity of these records, TPM is employed to record the states of all files.

3.4 SCOBA Procedure

The SCOBA procedure consists of three phases in our scheme: trusted building phase (trusted building in Figure 1), certification phase (step a, step b in Figure 1) and attestation phase (step c, step d, step e in Figure 1). TBS is responsible for the trusted building phase and records all required proofs for property certification on the target program. The trusted verifier issues the certificate according to these records generated by TBS. During the attestation phase, the challenger attests the target program with the help of the trusted verifier.

For a program P, its binary code files $F_e = \{f_{e_1}, f_{e_2}, ..., f_{e_i}\}$ is built from its corresponding source code files $S = \{f_{s_1}, f_{s_2}, ..., f_{s_j}\}$ with specified building configurations $C = \{c_{s_1}, c_{s_2}, ..., c_{s_j}\}$ and other dependent files $F_d = \{f_{d_1}, f_{d_2}, ..., f_{d_k}\}$, where f_e is an executable file of P, f_{s_i} denotes a source code file for P and c_{s_j} stands for the building configuration of f_{s_j}. These building configurations may be stored in some script files $F_C = \{f_{c_1}, f_{c_2}, ..., f_{c_m}\}$, such as Makefile, .config files on Linux and the building command options. Other dependent files include mainly library files used during the building process.

3.4.1 *Trusted Building Phase*

In the trusted building phase, we bind the binary code of a program with its source code and building configuration. By leveraging a secure domain and TPM, a trust chain is built from the source code and building configuration to the generated binary code.

In order to construct the trust chain from TPM, two PCRs are employed in our scheme: one for authenticated boot of TBS (PCR_{ab}) and another for the trusted building process(PCR_{tbp}). These two PCRs are reset at the initialization stage of the secure domain. When the attester starts the trusted building process, a secure domain is initiated by the Secure Virtual Machine and the subcomponents of TBS are measured and recorded with an authenticated boot before it is about to run. After TBS finishes initialization, it starts to compile the target source code files and records the state of input and output files. TBS employs TPM to record the compilation process with PCR_extend. All inputs, intermediate outputs and generated codes are recorded to attest the compilation process.

As shown in Figure 3, a typical compilation task is carried out in roughly five stages: Preprocessing, Parsing, Translation, Assembling, and Linking. We may consider each stage as a transformation process with certain inputs and outputs. As shown in Figure 4, we may consider the compilation process as a sequence of transformations. The output of each prior stage can be the input of the next stage. The output of each stage may be in different forms according to different compiler implementations and building configurations. Usually the output includes specific data structure in the compiling process, and other supporting files. The TBS records the states of these output files and binds it with its corresponding inputs.

A transformation process T may have input files $F_{in} = \{f_{in_1}, f_{in_2}, ..., f_{in_i}\}$ from the prior transformation process, output files $F_{out} = \{f_{o_1}, f_{o_2}, ..., f_{o_j}\}$ and other dependent files $F_d(T) = \{f_{d_1}, f_{d_2}, ..., f_{d_k}\}$. For example, the object files generated by the Assembling process are the inputs files of Linking stage; the executable files produced at the Linking stage (Ⓕ) are output files; the library files at the Linking stage are dependent files.

As shown in Figure 4, four key points for monitoring and recording the trusted building process are identified:

- Point Ⓢ:the moment immediately before the initialization of the building process ;
- Point Ⓐ:the moment immediately before a transformation process T is going to run;
- Point Ⓑ:the moment immediately after a transformation process finishes.
- Point Ⓕ:the moment immediately after the trusted building process terminates ;

At each point, the monitoring and recording actions are required to execute according to the following rules:

- Point Ⓢ: For all script files in F_C that store the building configuration, TBS measures and extends them with TPM: $H_{c_k} = SHA1(f_{c_k})$, where $SHA1$ stands for an SHA-1 hash function; $H_C = SHA1(H_{c_1}||H_{c_2}||...||H_{c_k})$; $PCR_{tbp}\_extend(H_C)$. For all source code files and corresponding configurations, TBS records their states and extends them into the TPM: $H_{s_i} = SHA1(f_{s_i}||c_{s_i})$; $H_S = SHA1(\ H_{s_1}\ ||\ H_{s_2}\ ||...||\ H_{s_i}\ )$; $PCR_{tbp}\_extend(H_S)$. If these building configurations are stored in some configure files, these files are also recorded and extended by the TPM.

- Point Ⓐ: At the beginning of a transformation process T, all files in F_{in} are recorded and extended by TPM: $H_{in_i} = SHA1(f_{in_i})$; $H_{in}(T) = SHA1(H_{in_1}|| H_{in_2}||...||H_{in_i})$; $PCR_{tbp}\_extend(H_{in}(T))$. All dependent files, if exist, are also recorded and extended by TPM: $H_{d_i} = SHA1(f_{d_i})$; $H_d(T) = SHA1(H_{d_1}||H_{d_2}|| ...||H_{d_i})$; $PCR_{tbp}\_extend(H_d(T))$.

- Point Ⓑ: TBS records the state of all output files and employs TPM to extend their measurements: $H_{o_i} = SHA1(f_{o_j})$; $H_{out}(T) = SHA1(H_{o_1}||\ H_{o_2}||...||\ H_{o_i})$; $PCR_{tbp}\_extend(H_{out}(T))$. For an output file f_{o_j}, all input files which determine the generation of f_{o_j} are also recorded in set: $F_{in}(f_{o_j}) = \{f_{in_{j1}}, f_{in_{j2}}, ..., f_{in_{ji}}| f_{in_{ji}} \in F_{in}\}$.

- Point Ⓕ: At the termination stage of compilation, TBS records the states of all output executable files: $H_{e_i} = SHA1(f_{e_j})$; $H_e = SHA1(H_{e_1}||\ H_{e_2}||...||H_{e_i})$; $PCR_{tbp}\_extend(H_e)$. At last, TBS will employ the TPM to generate a signature on the final values in PCRs:

$$Quote_{tbp} = sig\{PCR_{tbp}\}_{AIK_{priv}}$$
$$Quote_{ab} = sig\{PCR_{ab}\}_{AIK_{priv}}$$

  where AIK_{priv} is the private attestation key of TPM.

As the TPM extends all these records in sequence, an unbroken chain is established between the generated binary code and the source files with a given building configuration.

3.4.2 *Certification Phase*

After the trusted building process terminates, the attester sends a certificate request to a trusted verifier (step a) with the following messages:

$$\{F_e, S, F_C,\ H_e, H_S, H_C, H_{in}, H_d, H_{out}, PCR_{tbp}, PCR_{ab}, Quote_{tbp}, Quote_{ab}, AIK_{pub}, cert\{AIK_{pub}\}, SIG_M\}$$

where SML stands for Stored Measurement Log, AIK_{pub} stands for the public attestation key of TPM, $cert\{AIK_{pub}\}$ means the trusted certificate of TPM, $H_{in} = \{H_{in}(T_1), H_{in}(T_2), ..., H_{in}(T_i)\}$ is the set of input file records for all transformation processes, $H_d = \{H_d(T_1), H_d(T_2), ..., H_d(T_i)\}$ is the set of dependent file records for all transformation processes, SIG_M is the signature of these message which is generated with the session keys between the attester and trusted verifier. We assume that the communications between the attester and trusted verifier are protected. When the original provider of the customized software plays as the trusted verifier, it is not necessary to send all source code

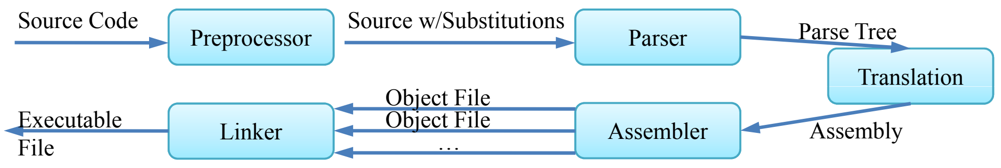

Figure 3: A typical compilation process in a trusted building phase.

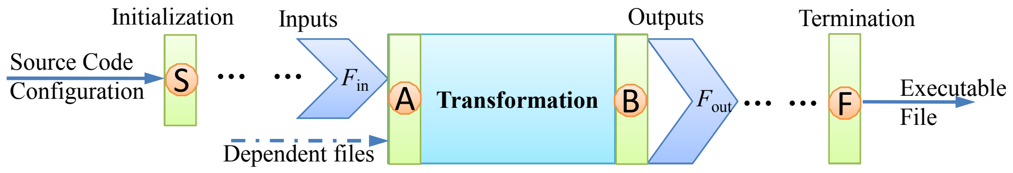

Figure 4: The detailed steps in a transformation process.

and configuration back to the trusted verifier. Only an update based on a standard version is required, such as the case with the Linux kernel patch.

The trusted verifier can conclude with certain properties for P by examining the received messages with following steps:

- First, the trusted verifier needs to attest the validation of TPM by checking its certification $cert\{AIK_{pub}\}$ and verify the integrity of messages.

- Second, it verifies the authenticated boot of TBS by checking PCR_{ab} and $Quote_{ab}$.

- Third, the verifier validates the trusted building process by checking $PCR_{tbp}, Quote_{tbp}$ according to F_e, S, H_C, H_e, H_S, H_{in}, H_d, H_{out}. Specially, the integrity of intermediate output files are required to be checked. For a input file f_{in_i}'s record H_{in_i} in transformation process (T_{i+1}), H_{in_i} should be consistent with its record H_{o_j} as output file in T_i.

- Finally, with all prior steps succeeded, the verifier will examine the source code, building configurations and binary code to determine the property of P. We will introduce a semantic approach of certifying the building configurations and source code in **Section 3.5**. If any of the above steps fails, the certification phase terminates with failure.

With a successful result, the trusted verifier issues the certificate on P and returns it to the attester platform (step b in Figure 1):

$$cert(TV, p, H_e) = \\ (H_e, H_S, H_d, H_C, p, sig\{H_e||H_S||H_d||H_C||p\}_{SK_{TV}})$$

where (PK_{TV}, SK_{TV}) is the key pair of trusted verifier TV for signature, p is a property, $cert(TV,\ p,\ H_e)$ denotes the property certificate for P. With the certificate $cert(TV,\ p,\ H_e)$, the trust chain is linked from the source code to the properties of generated binary code via a trusted building process.

3.4.3 Attestation Phase

In the attestation phase, we assume the attestation target program is P. The challenger first sends a challenge request to the attester with a nonce (step c in Figure 1). Then the attester platform employs the attestation agent to collect the state and certificate of P. These records are sent back to the challenger as the challenge response (step d in Figure 1). Then the challenger may verify the certificate with the help of the trusted verifier and concludes with an attestation result on P(step e in Figure 1). During the verification stage in the attestation phases, challenger only has to submit the collected certificates to the trusted verifier to verify its validation, and with a successful verification result, the challenger can verify the runtime measurements according to these certificates.

3.5 Property Certifying via Semantic Verification on Building Configuration

During the certification phase, the trusted verifier needs to certify the program by examining the source code and building configuration to judge the property of the generated binary code. The property of the binary code is strictly linked with the building configuration. Let's take Gentoo Linux [9] as an example: Gentoo Linux employs Use Variable Descriptions (Global/Local Use Flags) to indicate which software features are included, and finally generates packages with different properties. Meanwhile, the source code of different versions hold different properties. The trusted verifier maintains a database for recording the properties according to their source code versions and building configurations. The trusted verifier will use this database to check the received source code and building configurations to determine the property of generated binary code.

It is straightforward to manually examine the building configuration according to standard configurations and limitations. However, it involves a lot of unnecessary effort, and it may involve human faults when it comes to a large number of configurations. With the cryptographic hash functions, it is also possible to automatically examine the hash values of the building configuration files, when only limited and predictable configurations exist according to involved program properties. However, the building configurations may have huge amount of possible candidates and sometimes even infinite. When an option can be set as a floating value, the number of hash values for possible configuration files are almost infinite.

Fortunately, the building configurations are usually organized in a well-defined form, such as the Makefile, command options and .config files. Thus it is practical to examine the building configurations in a *semantic* way. We may consider a building configuration file as a collection of option pairs $< option, value >$. The trusted verifier has a set of criteria items $< option, criteria, operation, p >$ according to a specific program property p. The *operation* is determined by the type of option *value*. For example, the possible operations for integer values or floating values can be *equal*,*unequal*,*smallerthan*, etc. The operation set for all criteria can be determined according to the syntax of configuration file. In order to check whether the building configurations satisfy a certain property, an automatic process can be carried out to compare the option *value* with corresponding *criteria* according to specified *operation*. When the criteria of a specific program property is satisfied, the trusted verifier can conclude that the building configurations are with the property.

To perform semantic attestation on software configurations, we can apply a similar approach recently proposed in [24].

3.6 Property Certificate Revocation

It is possible that a program P, which is built based on a specific version of source code and building configuration, may be later found to be vulnerable or erroneous. So the trusted verifier needs to maintain a certificate revocation list to be able to revoke the corresponding property certificate. Once a program is identified as vulnerable, the corresponding certificate is added into the revocation list. During the attestation phase, the challenger is required to first check whether the property certificate is in the revocation list at certificate verification step (step e in Figure 1).

4. SCOBA IMPLEMENTATION FOR OPEN-SOURCE SOFTWARE

The proposed SCOBA framework could be applied to general custom software, however, it is most suitable for custom open-source software, where automatic attestation could be provided based on the open-source distributions. In this section, we apply SCOBA specifically to deal with custom open-source software, where users are allowed to tailor and configure the downloaded software, but are not allowed to modify the specific source code files. For cases of modifying source code, we will discuss it in Section 7.

We implement a prototype of this SCOBA framework to demonstrate its practical usage. Particularly, we focus on the customized open-source software on the Linux platform.

4.1 Attester Platform

We employ XEN [3] supported by the TXT facility as the Secure Virtual Machine. Ubuntu Linux is chosen as the operating system to host our prototype. In our implementation, we use Linux to run in two different domains of XEN: one is for ordinary applications in the ordinary domain of XEN and another is for the Trusted Building System in a protected domain. For the ordinary one, we introduce the attestation agent module as a Linux Security Module to monitor and record the execution of applications. For the TBS, we configure the Linux kernel via Linux From Scratch to get a minimal kernel to support the compilation tools, which carries out a trusted building process. We employ TXT to dynamically set up the secure domain for TBS [5].

4.2 Trusted Verifier

Trusted verifier maintains following repositories: a repository of known-good source code files and building configurations, a certificate repository, and a revocation list.

The known-good repository helps the trusted verifier to certify customized software. The known-good repository also records the properties of a software with specific source code files and building configurations for certain versions. The trusted verifier can automatically obtain the property of the target customized software. The certificate repository holds the records of all issued certificates and revocation list. The trusted verifier employs it to finish the certificate verification.

4.3 Trusted Building System

The Trusted Building System is the core of our scheme, and we will study its implementation based on GCC (GNU project C and C++ compiler) on Linux.

The GCC compilation process normally involves four steps: preprocessing, compiling, assembling and linking. The preprocessing step usually does not involve intermediate outputs, so TBS only has to monitor the intermediate outputs of following steps: compiling, assembling and linking. At the beginning of the above steps, we insert hooks into *gcc*, *as* and *ld* to monitor the inputs and outputs of these transformation processes. These hooks employ TPM to record the states of these inputs and outputs, and extend these records with the PCR_extend operation. At the end of the compilation process, TPM is invoked to generate quotes on these recorded proofs. In order to counter the "Time-of-measurement and Time-of-use" issue, we employ a similar mechanism as IMA [21] to deal with this problem.

4.4 Evaluation

We evaluate our prototype of TBS on a Lenovo ThinkPad X60 laptop with Intel Core 2 CPU T5500 @ 1.66GHz, and 1GB memory. We build a number of open-source applications with and without the proposed prototype, and the performance comparison is shown in Table 1.

In the table, we show the number for source code files, compilation time before and after applying the proposed scheme for each application. The cost for recording these proofs are roughly proportional to the number of source files in each application. The results show that TBS incurs roughly 2-4X slowdown on the evaluated benchmarks. The exception is TPM tools, which has an overhead of almost 15X because it involves only a handful of source files, thus the compilation time is relatively very short.

The overhead is pretty significant because of the large amount of TPM extend operations and low computation capability of TPM. However, the cost is still acceptable in practice, as TBS is only executed once for each build immediately before the certification.

Table 1: Comparison of compilation time before and after applying the proposed scheme.

applications	# of source code files	GCC-4.4.2	GCC-4.4.2 with TBS hooks
TPM-tools-1.3.4	59	14402 ms	209814 ms
Openssl-0.9.8k	1267	158106 ms	1318902 ms
Gmp-4.3.0	898	160279 ms	646499 ms
Trousers-0.3.1	326	118463 ms	345175 ms
Tboot-20090330	429	133646 ms	405173 ms
Linux-2.6.30	23214	7007143 ms	29034100 ms

5. CASE STUDY: APPLYING SCOBA TO GENTOO

Gentoo [9] is a free operating system based on either Linux or FreeBSD that can be automatically optimized and customized for just about any application. Most applications are distributed in the form of source code in Gentoo and its package management tool Portage is responsible for building and installing these applications. We can apply SCOBA straightforwardly to Portage to support attestation on custom software in Gentoo systems. Besides our modified GCC compilation tools with TBS hooks, we may also leverage Portage to provide a more flexible monitoring and recording mechanism for attesting customized software in Gentoo.

Portage is the heart of Gentoo, and performs many key functions. It serves as the software distribution system for Gentoo. It can maintain a local Portage tree which contains a complete collection of scripts that can be used by Portage to create and install the latest Gentoo packages. Portage is also a package building and installation system. It will build a custom version of the package to the user's exact specifications, optimizing it for the hardware and ensuring that only the optional features in the package that the users want are enabled.

Portage is characterized by its main function: compiling from the source code of these packages that the user installs. In doing so it allows customizing package functionalities to the user's own wishes, and customizing all packages to the systems specifications. In order to accomplish this, several functionalities are provided. Functionalities concerning managing the system are: allowing parallel package version installation, influencing cross-package functionalities, managing an installed-packages database, providing a local ebuild (explained later) repository, and syncing of the local Portage tree with remote repositories. Functionalities concerning installing the individual package are: specifying compilation settings for the target machine, and influencing specified package components.

The basis for the entire Portage system is the *ebuild* scripts. They contain all the information required to download, unpack, compile and install a set of sources, as well as how to perform any optional pre/post install/removal or configuration steps. An ebuild is a specialized bash script format created by the Gentoo Linux project for use in its Portage software management system, which automates compilation and installation procedures for software packages. Each version of an application or package in the Portage repository has a specific ebuild script written for it. The script is used by the *emerge* tool, also created by the Gentoo Linux project, to calculate any dependencies of the desired software installation, download the required files (and patch them, if necessary), configure the package, compile, and perform a sandboxed installation. Upon successful completion of these steps, the installed files are merged into the live system, outside the sandbox.

Base on the characteristics of Gentoo, we can easily extend TBS into the Gentoo Portage, and hence support trusted building in Gentoo. There are a number of different functions that we can define in ebuild files that control the building and installation process of the package. Hence, we can add specific TBS hooks in the call-sites of these functions in Portage to perform monitoring on the trusted building and installing procedure. These functions include:

- *Pkg_setup*: This function can perform any miscellaneous prerequisite tasks. This might include checking for an existing configuration file. We can add functions to initialize a trusted and isolated environment for the building procedure.
- *Src_unpack*: This function unpacks the sources, applies patches, and runs auxiliary programs such as the autotools. We can initialize the trusted measurement repository for all the source codes. Normally, the source codes are distributed in a single compressed package (e.g. tar file). Hence we should first generate the genuine measurement value for each file in the package (e.g. source codes, configuration files, etc.) from the signed measurement value of the source code package.
- *Src_compile*: This function configures and builds the package. We can integrate our trusted building mechanisms here.

Moreover, the following functions can be modified for implementing advanced trusted installation procedures, e.g. generating proof chains or related certificates.

- *pkg_preinst*: The commands in this function are run just prior to merging a package image into the file system.
- *Src_install*: This function installs the package to the destination.
- *Pkg_config*: This function sets up an initial configuration for the package after it's installed.
- *Pkg_postinst*: The commands in this function are executed immediately following merging a package image into the file system.

The package repository of Gentoo is in the best position to serve as the trusted verifier. Besides the package data, the package repository also maintains the corresponding property information in order to certify customized software. In order to support runtime certificate verification, the package repository maintains the certificate repository and revocation list.

6. RELATED WORK

Since TCG attestation was introduced as a key feature in the TCG specification[23], many remote attestation schemes have been proposed in the literature. Terra [7] employs a Trusted Virtual Machine Monitor (TVMM) to transform a tamper resistant hardware platform into multiple isolated virtual machines (VMs). With the protection of the trusted hardware, TVMM offers both the open-box VM and the closed-box VM. The attestation in TVMM only measures the programs before their executions and is not able to check their behaviors after attestation. As an extension of TCG attestation, IMA [21] employs a loading time integrity measurement mechanism which measures all software components including BIOS, the OS loader, the operating system, and programs at the application layer. The limitation of integrity-based attestation such as IMA is that it checks at the loading time. Since there exists a gap between time of measurement and time of execution, loading time integrity does not necessarily lead to stronger security assurance. As a follow-up of IMA, [20] employs IMA to enforce remote access control by attestation.

Property-based attestation [4, 19, 18] was introduced to provide a scalable attestation framework to support privacy preserving for the attester platform. A trusted third party is introduced to exam the runtime measurements and judge the property of the target platform. The challenger only verifies the property certificate to conclude the attestation result and the configuration information of the attested platform is preserved. Existing schemes of TCG attestation and property-based attestation are based on the known-good measurements of these attested programs.

Haldar et al. [12] introduced a semantic attestation mechanism based on the Trusted Virtual Machine (TVM). The TVM based semantic attestation mechanism enables the remote attestation of high-level program properties. Shi et al. proposed a fine-grained attestation scheme called BIND [22]. It provides evaluation interfaces to attest the security-concerned segments of code. Jaeger et al. [14] introduced the Policy-Reduced Integrity Measurement Architecture (PRIMA) based on the information flow integrity checking against the Mandatory Access Control (MAC) policies. Program execution attestation introduced in [11] is to attest whether a program is executed as expected. These semantic attestation mechanisms still require a know-good binary code repository.

However, most of the existing schemes are still based on binary attestation, as it plays an important role for authentication on software. As the binary attestation involves verification on the measurement of binary code, most of existing schemes have to face the problem of keeping a huge known-good measurements database in practical solutions.

Trusted Execution Technology (TXT) and Secure Virtual Machine (SVM) are introduced to provide a trusted execution environment. Recent years, there are already several practices [8, 15, 17] exploiting TXT or SVM. Open Secure LOader (OSLO) [15] leverages the dynamic root of trust to implement a bootloader based on AMD *SKINIT* instruction. Flicker [17] was introduced as an infrastructure for executing security sensitive code in complete isolation. It leverages the Secure Virtual Machine (SVM) of AMD processors and provides fine-grained attestation on program execution. LaLa [8] combines the latest hardware virtualization and trust technologies to deliver a more robust platform to support both instant-on system and a full-featured OS, and the flexible architecture enables a platform user to benefit from the advantages of a fast booting platform and a full-featured mainstream OS at the same time.

7. DISCUSSION

The proposed SCOBA framework could be applied to general custom software provided that a trusted verifier could be provided for all source files and configurations, which is not always practical. Here we discuss some of the limitations and possible enhancements of the proposed approach.

Selection of Trusted Verifier

It is important to choose the right party to play as the role of trusted verifier. In order to certify a customized software, the trusted verifier is supposed to have enough knowledge for carrying out the certification process. The provider of the original software holds the best position to serve as the role of trusted verifier for certifying the property of the customized software. However, when the original provider is not trusted or not available, a trusted third party can be employed and it should maintain a repository to store the property information of all known-good source code, which may come from different software providers, another trusted third party or trusted agent for software certification.

Automatic source code certification on custom-built software

For a custom-built program with only variant building configurations, the trusted verifier can employ semantic verification to automatically examine the building configurations. If the custom-built software does not make any modifications on the source code, the trusted verifier can maintain a repository of known-good source code files according to specific properties. In the certification phase, the proofs of trusted building process for the target custom software can be automatically analyzed to conclude its property.

Attestation on fully custom software

For fully custom software, users may modify the source code of the target custom software or even add new source code files into the software. It is difficult for a trusted verifier to automatically certify the modified source code. A straightforward way is to have experts manually checking these modifications and determine the property of the custom software. For programs with source code modifications at lower granularity (such as instructions), besides the manual verification on these modified codes, the trusted verifier can also employ more sophisticated certification techniques for automatic program certification, such as testing [6] and model checking. The certification on a whole customized software can be accomplished by certifying its software components [10]. The custom software may be built from scratch, and its source code files or subcomponents may come from other open source software. So it is possible to automatically certify these subcomponents from known software distributions.

Supporting semantic based attestation on custom software

The proposed scheme can serve as a building block for other types of semantic based attestation [12] on customized software. Different types of semantic attestation solutions may concern different properties of software. However, the integrity of a program is the basis for all different solutions. Our scheme provides the possibility to attest the customized software with unpredictable versions and configurations.

8. CONCLUDING REMARKS

In this paper, we introduce SCOBA, a source code based attestation scheme for custom software. SCOBA enables property attestation on custom software with unpredictable versions and building configurations. With a trusted building process, SCOBA binds the binary code of a program with its source code and building configuration. Then a trusted verifier is able to certify the generated binary code with the proofs from the Trusted Building System and determine the property of the target custom software by checking the source code and building configurations. Thus SCOBA links the trust chain between TPM to the runtime attested custom software. We implement a prototype of SCOBA based on GCC compilation tools and TPM. Experiments show that the performance is acceptable in practice. We also studies the application of SCOBA on Gentoo to support attestation on free software distributed in the source code form. With the support of SCOBA, it is possible for the free software community to employ remote attestation, one of the key TCG feature, to support trust establishment on applications in an open networking environment.

9. REFERENCES

[1] Linux From Scratch. `http://www.linuxfromscratch.org/index.html`.

[2] AMD. AMD64 Virtualization Codenamed "Pacifica" Technology–Secure Virtual Machine Architecture Reference Manual. Technical Report Publication Number 33047, Revision 3.01, AMD, May 2005.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, volume 37, 5 of *Operating Systems Review*, pages 164–177, Oct. 19–22 2003.

[4] L. Chen, R. Landfermann, H. Löhr, M. Rohe, A.-R. Sadeghi, and C. Stüble. A protocol for property-based attestation. In *STC '06*, pages 7–16, New York, NY, USA, 2006. ACM Press.

[5] J. Cihula. Trusted Boot: Verifying the Xen Launch. `http://www.linuxfromscratch.org/index.html`. Xen Summit 07 Fall.

[6] G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. In *ACM SIGSOFT Software Engineering Notes*, volume 22(4), 1997.

[7] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra a virtual machine-based platform for trusted computing. In *SOSP 2003*, Bolton Landing, New York, USA, October, 2003.

[8] C. Gebhardt and C. Dalton. Lala: a late launch application. In *STC '09: Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 1–8, New York, NY, USA, 2009. ACM.

[9] Gentoo. Gentoo Linux. http://www.gentoo.org/, 2009.

[10] A. K. Ghosh and G. McGraw. An approach for certifying security in software components. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 42–48, 1998.

[11] L. Gu, X. Ding, R. H. Deng, B. Xie, and H. Mei. Remote attestation on program execution. In S. Xu, C. Nita-Rotaru, and J.-P. Seifert, editors, *STC*, pages 11–20. ACM, 2008.

[12] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation—a virtual machine directed approach to trusted computing. In *the Third virtual Machine Research and Technology Symposium (VM '04). USENIX.*, 2004.

[13] Intel Corporation. Intel trusted execution technology — preliminary architecture specification. Technical Report Document Number: 31516803, Intel Corporation, 2006. `ftp://download.intel.com/technology/security/downloads/31516803.pdf`.

[14] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *SACMAT '06*, pages 19–28, 2006.

[15] B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of the 16th USENIX Security Symposium*, 2008.

[16] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[17] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. In J. S. Sventek and S. Hand, editors, *EuroSys*, pages 315–328. ACM, 2008.

[18] J. Poritz, M. Schunter, E. Van Herreweghen, and M. Waidner. Property attestation—scalable and privacy-friendly security assessment of peer computers. Technical Report RZ 3548, IBM Research, May 2004.

[19] A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. *New security paradigms*, 2004.

[20] R. Sailer, T. Jaeger, X. Zhang, and L. v. Doorn. Attestation-based policy enforcement for remote access. In *CCS 04*, October 25-29, 2004.

[21] R. Sailer, X. Zhang, T. Jaeger, and L. v. Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, USA, August, 2004.

[22] E. Shi, A. Perrig, and L. V. Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *2005 IEEE Symposium on Security and Privacy*, 2005.

[23] Trusted Computing Group. TPM main specification. Main Specification Version 1.2 rev. 85, Trusted Computing Group, Feb. 2005.

[24] H. Wang, Y. Guo, and X. Chen. Saconf: Semantic attestation of software configurations. In *ATC '09: Proceedings of the 6th International Conference on Autonomic and Trusted Computing*, pages 120–133, 2009.