15-316: Software Foundations of Security and Privacy

Lecture Notes on Trusted Computing

Matt Fredrikson

Carnegie Mellon University
Lecture 19

1 Introduction

In the previous lecture we delved into trust, and saw how it is possible to leverage existing trust relationships into new ones. The key technique that enables this uses digital certificates and trusted certificate authorities, forming a Public Key Infrastructure (PKI) that manages the distribution of certificates and the keys needed to verify them. PKI is an essential component of modern distributed systems, as it facilitates secure communication and dependent functionality across disparate geographic and organizational boundaries. Without it, computing as we know it today would not exist.

In today's lecture, we will take a closer look at an important application of PKI called *trusted computing*. The goal of trusted computing is to allow distributed systems to extend trust relationships to the software and underlying platform (i.e., the operating system, device drivers, ...) of remote machines [1]. This is a challenge, because it is not always reasonable to assume that a remote system is free from compromise that would affect its expected behavior. For example, many companies allow employees to work remotely from laptops and mobile devices. Once an employee is authenticated, they should be able to access sensitive resources and commit changes. This introduces the risk of malicious parties doing the same from their own devices, or of employees' devices unwittingly hosting malware that could leak or destroy these resources.

Trusted computing provides a way to verify that the software running on an a remote system is exactly what one expects for secure functionality—e.g., the correct operating system and tools, without surreptitious malware running on the side. Additionally, trusted computing allows the company to verify that the physical machine used to authenticate into the network is in fact owned by the company, and has not been subverted or modified. The key components that enable this are *trusted, tamper-proof hardware* and *remote attestation*.

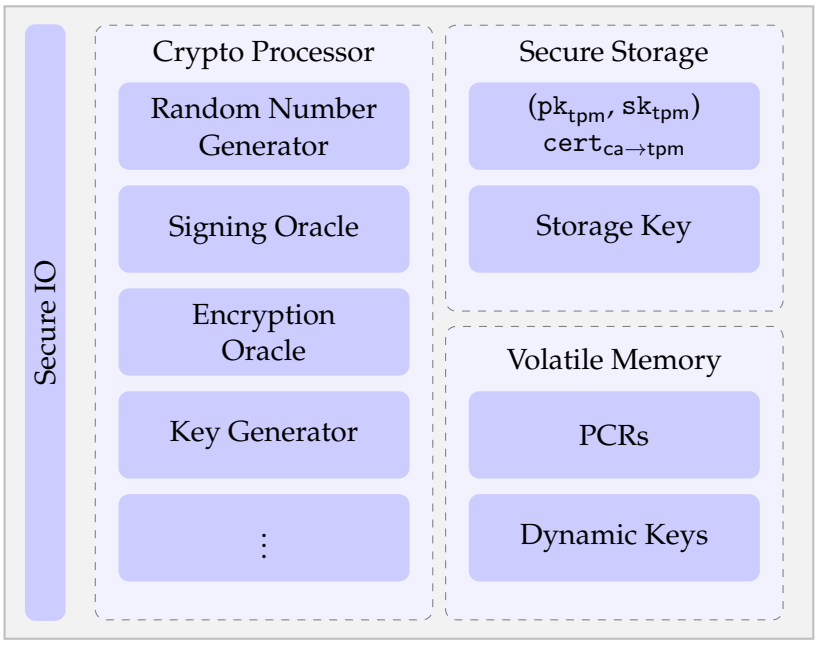


Figure 1: Summary of Trusted Platform Module components. All access to the TPM is mediated by a secure input/output layer to maintain the integrity of protected contents and to ensure that only authorized code can utilize trusted services.

2 Trusted Hardware

One of the main problems to address in realizing trusted computing is establishing a "root of trust" on a given system. In the previous lecture, we saw how certificate authorities can be viewed as a root of trust from which other trust relationships are extended. However, to do this we had to assume that the CA's public key is known to any party who verifies a certificate, and that only the CA is able to us its private key to issue certificates. If we want to be able to interact with a remote system, whose hardware we cannot directly inspect, then we need some way of establishing trust in statements made by the system regarding its configuration and software.

One idea is to store a secret key that is unique to the system, perhaps within the operating system. That way, the operating system could inspect the code running on the machine when needed, and sign a corresponding statement with the secret key. However, for others to trust these statements, they would also then need to trust that the operating system itself has not been compromised and had not leaked its signing key. In principle this might seem fine, but in practice this is a rather strong assumption to make because conventional OSes are large, buggy pieces of software that get compromised routinely.

In short, adding the OS to the *trusted computing base* is a bad idea because it is too large and therefore difficult to account for. Perhaps instead we could push the secret key

further down in the platform stack, perhaps in the BIOS or as a priveleged instruction implemented by the CPU itself. To mitigate the risk of leaking the signing key, we can expose a function that uses the key to sign messages (called a *signing oracle*), rather than exposing the key itself. However, typical BIOS functions and priveleged instructions can be invoked by the OS, and since we do not trust the OS, the signing oracle should only be accessible to certain pieces of known, trusted code running on the system.

This is essentially the idea behind the *Trusted Platform Module* (TPM). TPM is a standard for secure hardware co-processors that provide functionality for securely storing of small amounts of data, signing data with trusted keys, and measuring the state of the system (e.g., registers, memory, ...). The secure storage on the TPM comes pre-loaded with a unique key pair $(\mathtt{pk}_{\mathsf{tpm}}, \mathtt{sk}_{\mathsf{tpm}})$ as well as a certificate issued by a trusted CA $\mathtt{cert}_{\mathsf{ca}\to\mathsf{tpm}} \equiv \mathtt{sign}_{\mathtt{sk}_{\mathsf{ca}}}(\mathtt{isKey}(\mathsf{tpm}, \mathtt{pk}_{\mathsf{tpm}}))$. The volatile memory in the TPM is used to store keys that are generated dynamically as software makes requests, as well as a set of *Platform Configuration Registers* that are used to store hash values that reflect the code currently running on the system. Importantly, the TPM is implemented as a tamper-proof hardware component with strong separation from the software running on the system, so the contents of the secure storage remain secret and the contents of volatile memory cannot be modified by rogue software. An overview is shown in Figure 1.

2.1 Roots of trust

The fact that the TPM resides in hardware that cannot be tampered with by software or by physical means establishes a core "root of trust" on which TPM-based statements can be based. The architecture of the TPM breaks this down a bit further into several associated roots of trust that cover specific aspects of the TPM's responsibilities, defining the *root of trust for measurement* (RTM) and *root of trust for reporting* (RTR).

Root of trust for measurement. The RTM addresses the question, *can we trust the component that examined system state reflected in the report*? The RTM must initiate the process of *measuring* the current state of the system, and ensure that it only hands this responsibility off to another trustworthy component. The TPM itself has no visibility into the rest of the system, so this root of trust must necessarily reside outside of the trusted hardware, and it must run before any other untrusted components. If it ran *after* an untrusted component, then there would be no reason to believe that the untrusted component did not violate the integrity of the RTM.

In practice, one way to establish this root of trust is to place it in the BIOS boot block, which is called the *core* RTM. The BIOS boot block is the first piece of code that runs when a system is started, and it must be trusted to faithfully record a hash of itself and the code that it hands control of the system to after performing its normal functions. In this way, it begins a series of handoffs that can later be inspected to form a transitive chain of trust from the CRTM to the operating system and applications that are eventually loaded and run on the system.

Root of trust for reporting. The RTM establishes trust in the process responsible for measuring the current state of the system. But in order to create a report for remote principals that attests to the integrity of the system state, these measurements must be stored securely prior to reporting. In particular, once measurements have been taken it should not be possible for software running on the system to overwrite or erase them, as this would allow rogue software like malware to cover its tracks.

This component is implemented by the TPM, and relies heavily on what are called *Platform Configuration Registers* (PCRs). The TPM has several dozen PCRs, which are 20 bytes in length (the size of a hash), in its volatile memory. However, the PCRs cannot be written to directly by software running on the system. Rather, the only way to update a PCR is to *extend* it, as shown in Equation 1.

$$\texttt{extend}(n, D) := \mathsf{PCR}[n] \leftarrow H(\mathsf{PCR}[n] \| D) \tag{1}$$

In (1), n is the PCR register number, D is a digest (i.e., block of bytes representing current state), and H is a hash function. The state of the specified register is updated with the hash of its current state, concatenated with the given digest. In this way $\texttt{extend}(\cdot, c)$ reates a *hash chain* in each register, which can be easily verified by any party who knows the correct sequence of $\mathsf{PCR}[n]$ values, but is difficult to forge as long as H is a good cryptographic hash function.

To understand this in a bit more detail, consider a scenario where the intial value of $\mathsf{PCR}[1]$ was set to 0. Then suppose that it was extended twice, with $\texttt{extend}(1, D_1)$ and $\texttt{extend}(1, D_2)$.

$$\begin{aligned} &\text{After } \texttt{extend}(1, D_1), \quad \mathsf{PCR}[1] = H(D_1) \\ &\text{After } \texttt{extend}(1, D_2), \quad \mathsf{PCR}[1] = H(H(D_1) \| D_2) \end{aligned}$$

Then if an application reports that the state of the system was D_1 followed by D_2, one can easily verify this by running the hash function. But if a malicious party wishes instead to report that the state of the system was something else, say E_1 followed by E_2, then they need to solve for two equations.

$$\begin{aligned} &\text{Find } E_1 \text{ such that} \quad H(E_1) = H(D_1) \\ &\text{Find } E_2 \text{ such that} \quad H(H(E_1) \| E_2) = H(H(D_1) \| D_2) \end{aligned}$$

Hash functions for which this problem is computationally infeasible are called *preimage-resistant*, and all cryptographic hash functions satisfy this property.

3 Measuring system state

The two roots of trust lay the foundation for the TPM's primary functions, which is to provide trustworthy information that can be used to verify the state of the system. In the context of trusted platforms, this process is called *measurement*. There are two settings in which measurement routinely happens, namely at boot time and then later as needed by user applications.

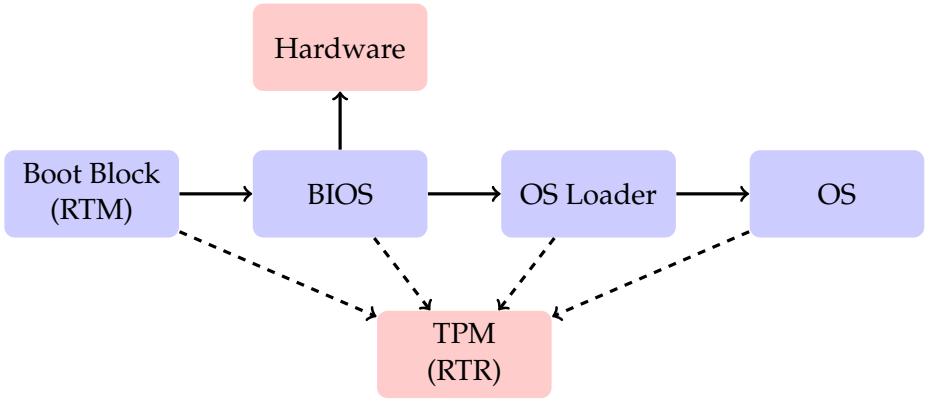


Figure 2: Authenticated boot process. Solid arrows denote measurements, and dashed arrows denote PCR extend operations. The root of trust for measurement is the BIOS boot block, and the root of trust for reporting is the TPM.

3.1 Authenticated boot

The fundamental idea behind authenticated boot is to establish that the final software platform, i.e. the operating system and its various components, was correctly loaded into memory and executed as expected by the user or system administrator. To accomplish this, each stage of the system's boot process must be carefully modified to make use of the RTR so that a secure record of each piece of code involved in setting up the system is maintained. This of course starts with the RTM, which records a digest of the next piece of code that runs and extends a PCR with it. Each bit of code continues in this manner, by measuring (i.e., computing a digest of) the *next* piece of code and extending a PCR.

This process is shown in Figure 2, with the typical stages of boot for commodity platforms. Because the BIOS boot block is the root of trust for measurement, it begins the process of measuring the next code to run. At each stage, the root of trust for reporting is utilized by extending the TPM's PCR registers with a digest summary of the next code immediately prior to handoff. When this finishes, one can verify that the code used to boot the system was trustworthy by inspecting the appropriate PCR, and checking its value against the expected digest of each stage's code.

Note that this process ultimately relies on each stage trusting the next to accurately report subsequent measurements. For example, if the BIOS had "hidden" functionality that recorded a measurement for malicious code, or even worse if the BIOS reported a measurement for the OS loader and then instead handed control to some malicious code, then the entire process has been subverted. Once the malicious code takes control, then it can extend the PCR with whatever values it chooses, and as long as the RTM remains trusted, then those values will be trusted.

Another important point about this process is that it assumes that whoever verifies the PCR contents is able to determine the suitability of the platform from the digest summaries. However, the digest will only depend on the actual bytes that comprise

the code running on the system, and say nothing about the semantics of that code. For example, a trusted party might implement an OS loader that contains vulnerabilities, which as we know can be difficult to identify. So even though the loader is trusted to do the right thing, it may be possible to subvert it while still passing PCR checks, because they say nothing about the behavior of the corresponding code.

3.2 Application integrity

Measurement does not need to end with the operating system, as there is no reason that this process shouldn't continue indefinitely as applications are executed on the system. Once it is established that the operating system is trusted, it can measure other applications' code and report the resulting PCR values to interested parties.

4 Using trusted measurements

The platform measurement process is the basis for most of the TPM's functionality, and can be used to implement a wide range of secure functionality. We will discuss two of the most important and widely-deployed examples, *remote attestation* and *sealed storage*.

4.1 Attestation

Once the system state is measured and recorded in secure PCRs, the process of reporting these measurements to remote systems is called *attestation*. Recall that the TPM has its own unique key pair $(\mathtt{pk}_{\mathsf{tpm}}, \mathtt{sk}_{\mathsf{tpm}})$ and a certificate $\mathtt{cert}_{\mathsf{ca}\rightarrow\mathsf{tpm}}$ vouching for the authenticity $\mathtt{pk}_{\mathsf{tpm}}$. It may seem that attesting to the state of the system at any point is as simple as using $\mathtt{sk}_{\mathsf{tpm}}$ to sign the contents of specified PCRs, and indeed for some purposes this may be just fine.

$$\mathtt{sign}_{\mathtt{sk}_{\mathsf{tpm}}}(\mathsf{PCR}[i] = \mathit{GoodHashVal}) \tag{2}$$

In particular, when used in combination with the TPM's CA-issued certificate, (2) would allow one to conclude that at some point since boot the state of the system was properly configured.

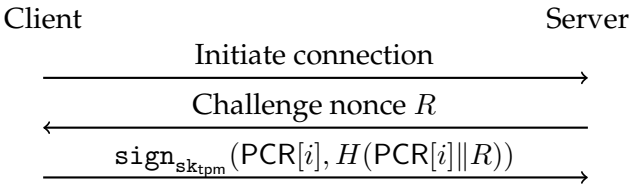
But the usefulness of this fact is rather limited, and many of the applications that trusted platform enables need more. In particular, remote systems often want to verify that they are communicating with an uncompromised host before giving them access to sensitive resources.

1. Client system completes authenticated boot, storing measurements in TPM's PCRs.
2. User (on client) wishes to use an application that consumes protected resources on remote server. User's application initiates a connection to the remote server, and sends a signed statement such as (2).
3. The remote server checks the signed PCR values and validates $\mathtt{pk}_{\mathsf{tpm}}$, in an attempt to establish that the client is in a secure state.

4. If successful, the server gives the client application access to the protected resources.

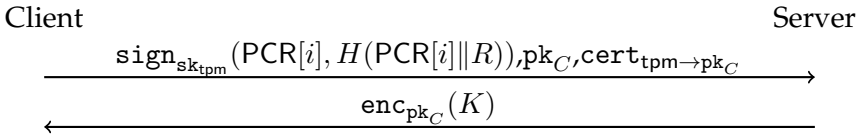
There are two big problems with this protocol. First of all, suppose that $GoodHashVal$ in (2) is a constant value that never changes when the user attempts to use the remote server. In other words, whenever the remote server attempts to validate the user's platform, it always checks the given PCR values against the same pre-defined constant. Then any adversary will know in advance what value the server checks for, and if they can obtain a statement signed by tpm to that effect then they can trick the server into believing that it is communicating with a secure client.

This is called a *replay attack*, and the protocol as outlined above is vulnerable to it whenever the client sends the server an attestation over an unsecure channel. All that the attacker needs to do is intercept this message once, and they will be able to violate the guarantees of trusted platform in subsequent connections. This attack can be thwarted by having the server reply initially with a *challenge nonce*, or hard-to-predict value that the client must incorporate into its attestation to establish the freshness of the session. This can be done by appending a hash of the PCR value with the nonce value, signed again with the TPM's secret key as shown below.



However, this protocol is still vulnerable to a *session hijacking* attack. After the client sends the server the signed attestation, the server will believe that any messages that come over the corresponding channel originated on a secure platform. If an attacker inserts themselves in the middle of the client and server, as with a Man-in-the-Middle attack, then they will be able to again fool the server into accepting rogue messages as trusted.

The only way to thwart the session hijacking attack is for the client and server to establish a *shared secret* that is used to encrypt all subsequent communications between the two. This utilizes additional functionality in the TPM, in particular the key generation routines in the cryptographic processor. After attesting with the server's nonce, the client asks the TPM to generate a new public, private key pair $(\mathtt{pk}_C, \mathtt{sk}_C)$, in addition to a certificate $\mathtt{cert}_{\mathsf{tpm}\to\mathtt{pk}_C}$ for the public key. It then sends the public key and cert to the server. The server likewise generates a secret symmetric key K, and uses the client's public key to encrypt and send it back. The client can decrypt the shared symmetric, which it uses to encrypt any subsequent messages to the server (and vice versa). The steps after the server's nonce are shown below.



Before moving on, it is worth noting some limitations of attestation as we have described it. First, while a successful attestation can give the remote party confidence that the correct software is running, it only describes the state of the system at the time that the application was loaded. As discussed previously, the PCR values say nothing about the correctness or behavior of the attested code, and by extension nothing about the state of the system after the measurement was taken. So for example, if the client application falls victim to a code execution vulnerability (e.g., buffer overflow or ROP) after being measured, then any trust that the remote party has in the client is misplaced because the process currently being executed on the client is not under their control.

Second, the trust assumptions that are leveraged in attestation are actually quite fragile. Surprisingly, if the secret signing key of even a single TPM is exposed to malicious parties, then the *entire* infrastructure for TPM-enabled attestations is compromised. This is because an attacker can use the compromised key to emulate a TPM, signing false PCR values for any attestation request that it wants. If that all TPM signing keys are endorsed by the same CA, then the endorsements originating from the attacker's false TPM are trusted due to a certificate issued by the same CA as any other properly-functioning TPM. Thus effective revocation procedures are crucial for the robustness of the entire TPM infrastructure.

4.2 Protected storage

Another important application of TPM measurements is protected storage, such as `dm-crypt` (Linux & BSD) and BitLocker (Windows). When a machine with TPM is first purchased and initialized, the owner can choose to create a *storage root key* (SRK) $\mathtt{sk}_{\mathsf{store}}$ that is stored on the TPM and only accessible after providing a user-defined password. The storage root key is then used to encrypt any number of secret keys $K_1, \ldots, K_n$ that are identified by unique *key handles*, which are used to provide *sealed storage* for particular applications on the system. This is accomplished via the $\mathtt{seal}(\cdot, \cdot, \cdot)$ function, which takes three arguments as shown in (3).

$$\mathtt{seal}(key\ handle, PCR\ values, data\ block) \tag{3}$$

The data block argument can contain up to 256 bytes of data, and is intended to contain an encryption key that is used to protect data for storage. `seal` returns a "blob" encrypted with the SRK-protected K_i corresponding to the given handle, which can only be decrypted by the TPM when the PCR values are consistent with those passed to `seal`. This allows a form of secure storage that requires the system to be in a particular configuration before data can be accessed.

The simplest form of sealed storage protects the entire operating system and all installed applications, and is accomplished by sealing the operating system on disk with PCR values that reflect the correct boot block, BIOS, and OS loader. If any of these change, then the PCR values at boot time will not match those with which the OS was sealed, and it will fail to load. Similarly, one can seal an application's code with PCR values that reflect the correct OS kernel, so that the application will only run if the

kernel has not been compromised. This can prevent attempts to violate the trusted computing base assumptions of the developers, or to stymie reverse engineering.

More generally, the ability to place constraints on PCR values for data access enables a wide range of policies relevant to secure software. One challenge faced by software that relies on secret encryption keys is, *how to store the keys so that they are not vulnerable to compromise of other parts of the system*? This is relevant, for instance, to secure HTTP servers that are exposed to untrusted network traffic. If the operating system or some other service becomes compromised, then the secret TLS keys can potentially be exfiltrated by the attacker. This could be addressed by having the server utilize sealing, so that the keys can only be decrypted by an unmodified version of the HTTP daemon. Another use could be to validate that certain conditions were met when a piece of data was signed, for example that sensitive patient records were obtained by an uncompromised database server.

5 Example: Network File Server

To wrap up, let's consider a more in-depth example of how trusted platform might enable secure functionality in a distributed environment. Suppose that your instructor mfredrik stores the gradebook for this course on a networked file server situated in some secure location on campus. Because student privacy is paramount, this server implements strict access controls while making very few assumptions about trust. The access control policy itself isn't particularly interesting, and the server daemon will simply attempt to verify from some assumptions (we will get to these in a minute) that it is indeed mfredrik who requests the gradebook.

$$\Gamma \vdash \text{mfredrik says } \texttt{read}(\text{'15316-grades.xlsx'}) \tag{4}$$

Because requests will always come from remote hosts, the daemon utilizes trusted platform to ensure that the request originates from a university-issued machine, running an up-to-date operating system that has authenticated mfredrik. So the server has been configured to trust the TPM CA's public key, embodied in policy Q_1.

$$Q_1 \equiv \texttt{isKey}(\texttt{ca}, \texttt{pk}_{\texttt{ca}}) \tag{5}$$

Having issued mfredrik his laptop, the university also knows the public key of his TPM, and has pre-cached its certificate.

$$Q_2 \equiv \texttt{sign}_{\texttt{sk}_{\texttt{ca}}}(\texttt{isKey}(\texttt{tpm}, \texttt{pk}_{\texttt{tpm}})) \tag{6}$$

When mfredrik powers his machine, it performs an authenticated boot and stores the measurements in the TPM's root of trust for reporting. When this completes, the operating system makes use of the TPM's secure random number generator to construct a public/secret key pair $(\texttt{pk}_{\texttt{os}}, \texttt{sk}_{\texttt{os}})$, and obtains a certificate from the TPM.

$$Q_3 \equiv \texttt{sign}_{\texttt{sk}_{\texttt{tpm}}}(\texttt{isKey}(\texttt{os}, \texttt{pk}_{\texttt{os}})) \tag{7}$$

Next mfredrik must authenticate to the operating system by providing his username and password. Because the operating system is known to the university, it is trusted to issue statements on mfredrik's behalf after he has successfully authenticated. In particular, the university trusts any requests to read files made by the operating system just as though they had been made directly by mfredrik himself.

$$Q_4 \equiv \texttt{sign}_{\texttt{sk}_{\texttt{os}}}(\forall x.(\texttt{os says read}(x)) \rightarrow (\texttt{mfredrik says read}(x))) \tag{8}$$

Q_4 in (8) may seem a bit strange at first glance. The operating system is essentially saying that it is allowed to make file requests on behalf of mfredrik. Why should the server trust the OS to claim authority for itself on behalf of other principals? We will assume that computing services has done its legwork to vet the operating system, and is confident that it would not make such a statement unless mfredrik had actually authenticated himself to the OS.

Finally, once logged on mfredrik issues a command to open up the gradebook. At this point, the OS signs a request to read the appropriate file (Q_5, and sends it to the networked file server along with the other policies (Q_3 and Q_4).

$$Q_5 \equiv \texttt{sign}_{\texttt{sk}_{\texttt{os}}}(\texttt{read}(\text{'15316-grades.xlsx'})) \tag{9}$$

On the other end, the file server has some work to do before it can decide to send the requested file back to the client. First it needs to be convinced that the software running on mfredrik's computer is free of compromise and up-to-date as expected. It initiates a remote attestation to check that the PCRs initialized by the authenticated boot match expected values, because if this is not the case then none of the vetting and trust placed in the operating system to make requests on mfredrik's behalf is warranted. Then it needs to verify that the OS's key was actually signed by the TPM, and not some rogue software on mfredrik's computer.

$$\text{Check } \texttt{verify}_{\texttt{pk}_{\texttt{tpm}}}(\texttt{sign}_{\texttt{sk}_{\texttt{tpm}}}(\texttt{isKey}(\texttt{os}, \texttt{pk}_{\texttt{os}}))) = \texttt{true} \tag{10}$$

Having established the integrity of the OS and the authenticity of its public key, it then checks to make sure that the subsequent statements Q_4, Q_5 actually came from the operating system and not some rogue software.

$$\text{Check } \texttt{verify}_{\texttt{pk}_{\texttt{os}}}(Q_4), \texttt{verify}_{\texttt{pk}_{\texttt{os}}}(Q_5) = \texttt{true} \tag{11}$$

If each of these steps is successful, then the file server can be confident that it is speaking to an uncompromised client, to which mfredrik has authenticated and issued a command to read the gradebook. Finally, it will attempt to prove the judgement in (4), and if successful send back the contents of the file. Constructing this proof is left as an exercise.

References

[1] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 1st edition, 2011.