Chapter 11

Measured Principals and Gating Functions

A program running on a processor can be considered a principal. Whether that principal will comply with a given security policy depends on what properties its executions satisfy. We might try to infer those properties by using the name of the program (e.g., Linux 4.8.0-36-generic or gcc 4.3.2) to identify code that we or some trusted authority then analyzes. But to make inferences about the principal’s executions from that analysis, we must also establish:

- the binary being executed corresponds to the code that was analyzed,
- initialization data read by the program is what the analysis assumed, and
- services that the execution environment provides perform as expected.

Measured principals provide a way to close that gap.

The defining characteristic of being a measured principal is having a name that is derived from the binary being executed, initialization data read, and the name of a measured principal that is providing the execution environment. Moreover, we require this name to be derived in such a way that changing one or more bits of those elements would lead to an unpredictably different name. The name of measured principal can therefore serve as a label for the properties we believe to be satisfied by the principal’s executions.

A gating function \([K-F](\cdot)\) is a cryptographic function obtained by adding access control that restricts uses of cryptographic function \(K\cdot F(\cdot)\) and its associated key \(K\):

- Key \(K\) can be used only by gating functions \([K-G](\cdot)\) naming \(K\). Other accesses to \(K\) are blocked.

\(^1\)The use of square brackets in “\([K-F]\)” is intended to indicate that access is restricted to \(K\) and \(F\). The notation also distinguishes gating functions from ordinary cryptographic functions.
Invocations of $K\cdot F(\cdot)$ can proceed only in systems satisfying configuration constraint $\text{Config}([K\cdot F])$, which is a set of measured principals that must have started executing.

So gating functions enforce a form of authentication-based access control, where binary executables as well as their run-time environments are being authenticated. And $\text{Config}([K\cdot F])$ provides assurance that invocations of $[K\cdot F](\cdot)$ will not violate a given security policy, because (i) an invocation of $[K\cdot F](\cdot)$ requires that the measured principals in $\text{Config}([K\cdot F])$ have started executing, and (ii) the name of a measured principal also identifies properties that we believe will be satisfied by that principal’s executions.

Gating functions support enforcement of a wide range of restrictions on access to digital content. Confidentiality can be enforced by using encryption. Integrity can be protected by using digital signatures, message authentication codes, or authenticated encryption—all make unauthorized updates detectable to subsequent readers, although unauthorized writes do compromise availability. Gating functions also can be used to generate attestations that provide information about the system configuration when a system response is produced.

Isolation is a combination of confidentiality and integrity. The isolation that gating functions and cryptography enforce differs from the isolation that the system abstractions—processes, virtual machines, and containers—that Chapter 10 describes. It is weaker in some ways and stronger in others. It is weaker because gating functions don’t protect availability, whereas the system abstractions block unauthorized writes and, therefore, do preserve availability. It is stronger because gating functions require that certain code be running as a prerequisite for access; in comparison, the system abstractions are agnostic about the code they are isolating and thus (inadvertently) might protect malware. Finally, attestation that gating functions can implement is not supported by the system abstractions.

### 11.1 Measured Principal Descriptions

One way to satisfy the requirements for a measured principal’s name $\mathcal{N}(D)$ is to derive that name from a description $D$, which is a sequence $d_1, d_2, \ldots, d_n$ of descriptors such that:

- changing one or more of the descriptors in $D$ results in a new description $D'$ with an unpredictably different name $\mathcal{N}(D')$.

---

2Using a set would be less expressive than using a sequence. For example, consider two measured principals: (i) an operating system that runs in a virtual machine, (ii) a hypervisor that is started by an operating system. Both measured principals involve the same resources (viz., a hypervisor and an operating system). A description defined by a set of descriptors for those resources would not distinguish between (i) and (ii); a description defined by a sequence, ordered by first access, does distinguish.

3For those wondering about the etymology of the term “measured principal”, some authors use the term measurement for what we call a descriptor and measurement list or measurement chain for what we call a description.
11.1. Measured Principal Descriptions

- each descriptor \( d_i \) is derived from some resource at the time of first access by the measured principal being named, and
- descriptors are listed in order of first access by the measured principal being named.

Examples of resources include: hardware processors, input/output devices, executables, regions of storage, and files. Our objective is for a description to indicate whether the associated measured principal can be trusted to satisfy some properties. This objective is facilitated if each descriptor can be interpreted as a label for properties satisfied by an associated resource.

A principal’s executions are determined by the processor on which it runs, all of the code that processor has executed since the last reboot (assuming reboot resets the processor state), and information read at initialization and subsequently. So a conservative implementation of the description for a measured principal \( P \) would include a prefix that comprises descriptors for all interpreters (hardware processors and/or software) and all storage (registers, memory, disk) accessed prior to starting \( P \). For example, in a description \( d_1 d_2 \ldots d_n \) for the measured principal associated with an operating system, \( d_1 \) might identify the hardware processor on which the operating system booted, \( d_2 \) might be derived from the firmware, \( d_3 \) might identify the contents of disk blocks comprising boot code, and so on.

Completeness of Descriptions. A description that does not include a descriptor for each resource that a measured principal has accessed could lead to inaccurate predictions about properties of executions. We then might trust a system that is not trustworthy or be prompted to install defenses against problematic executions that cannot actually arise. For example, because firmware and boot routines restrict possible execution by an operating system and its clients, a description for an operating system will be more useful for predicting possible executions if that description includes descriptors for the hardware processor, firmware, and boot routines.

More-complete descriptions bring other benefits, too. A more-complete description for a measured principal \( P \) can block attacks embedded in a modified version \( P' \) that masquerades as \( P \). This is because the additional code or file(s) accessed to create \( P' \) would produce a different description and, therefore, \( P' \) will be given an unpredictably different name from the name that \( P \) was given. The new name would cause gating functions to block \( P' \) from using the cryptographic keys that \( P \) is using to protect its content and authenticate its responses. So the attempt by \( P' \) to masquerade as \( P \) is foiled. Moreover, the unpredictability of the name \( P' \) is given means (with high probability) that name would be different from the name of any of the executing measured principals and, therefore, \( P' \) could not masquerade as any of those, either.

More-complete descriptions also can prevent attackers from achieving persistence by modifying files that will be reloaded each time the system restarts. It suffices that descriptions include descriptors for those files—descriptors that
(because the descriptor for a file will be calculated from that file's contents) would be different for the altered versions of the files. So, after the restart, a measured principal that accesses the corrupted files has a different description and, therefore, would be given an unpredictably different name from the name given to the measured principal that read the uncorrupted system files. Gating functions that required the old name would not work when invoked by a measured principal with the new name; the system restart would be unable to proceed.

Complete descriptions of execution environments can be a source of inconvenience, though. Customizations, patches, and upgrades change files contents. Because descriptors for files are derived from file contents, measured principals would now be given different names and their access requests would be denied unless some sort of upgrade protocol is run. Such a protocol ought to require that changes to the system come with a digitally-signed attestation that allows the system to verify the integrity of the updates; an approach like that sketched on page 315 might then be employed to change content being protected by gating functions.

### 11.2 Naming Schemes for Measured Principals

An attacker could co-opt a gating function $\[K-F](\cdot)$ if that attacker is able to run a measured principal that will be given the same name as a measured principal already allowed access by $\text{Config}([K-F])$. So there is good reason to prevent attackers from exerting control over the names that will be assigned to measured principals.

An attacker can control the order in which resources are accessed by a program it writes and runs, so an attacker can control the description that will be constructed for the resulting measured principal. But exerting control over a description $D$ for a measured principal does not suffice for co-opting a gating function if it is infeasible to predict name $N(D)$ that will be assigned to that measured principal. The following requirements on naming function $N(\cdot)$ provide that defense.

**Names for Measured Principals.** Function $N(\cdot)$ for assigning name $N(D)$ to a measured principal having description $D$ satisfies:

- **Collision Resistance.** If $D \neq D'$ holds then, with high probability, $N(D) \neq N(D')$ holds.\(^4\)

- **Preimage Resistance.** Given $D$, it is infeasible to construct a description $D'$ where $D \neq D'$ and $N(D) = N(D')$ hold.

\(^4\)Names will be fixed length and relatively short, whereas descriptions are neither. So there are more possible descriptions than names, and mapping $N(\cdot)$ from descriptions to names must map two or more different descriptions to the same name. But if the space of names is large, then the probability of such collisions would be low in the small set of descriptions that arise while a system is running.
Collision Resistance implies that, in practice, names of measured principals in a system will be unique. Therefore, it is unlikely that one measured principal would be authorized by some gating function just because that measured principal got assigned the same name as some other measured principal that already is authorized. Preimage Resistance further stipulates that it is infeasible for attackers to orchestrate one of those (unlikely) name collisions with a given name \( N(D) \) by creating a principal with description \( D' \) that satisfies \( N(D') = N(D) \).

Collision Resistance and Preimage Resistance also happen to be the defining properties of a cryptographic hash function \( H(\cdot) \). Therefore, the requirements for names of measured principals are satisfied if function \( N(\cdot) \) is defined in terms of cryptographic hashes.

By unwinding the recursion in (11.1), we see that \( N(d_1 d_2 \ldots d_n) \) incorporates the descriptors of \( \bullet d_1 d_2 \ldots d_n \) in order of appearance. So there is a function

\[
\mathcal{X}(N, d): \quad \mathbb{H}(N \cdot \mathbb{H}(d))
\]

(11.2)

to generate the name for a description \( \bullet d_1 d_2 \ldots d_n d \bullet \) that extends description \( \bullet d_1 d_2 \ldots d_n \bullet \) having name \( N \); we thus say that \( \mathcal{X}(N, d) \) extends name \( N \) with descriptor \( d \).

Allowing \( N(D) \) to be calculated incrementally accommodates descriptions that incorporate descriptors for files that have been read but become inaccessible while the description continues to evolve. System boot and other parts of startup typically access such files. Incremental calculation of \( N(D) \) also allows a descriptor \( d_{i+1} \) to become known only after descriptor \( d_i \) has been incorporated into the description—an important feature if \( d_{i+1} \) is not known until after \( d_i \) starts executing because, for example, \( d_i \) selects and loads its successor. Again, this situation often arises during system startup.

Names for measured principals must come from trusted sources. If that source is the local runtime environment and names are computed from descriptions according to (11.1), then it suffices for the integrity of a measured principal’s name—but not its description—to be protected.

**Trust Assumption for Measured Principal Names.** To trust name \( N_P \) for a measured principal, the only change allowed to \( N_P \) is the replacement of its value with \( \mathcal{X}(N_P, d) \) when a resource with descriptor \( d \) is first accessed by this measured principal.

Unlike names, descriptions for measured principals need not be protected because a simple check is available to determine if the integrity of a candidate description \( D_P \) has been compromised: if \( N(D_P) = N_P \) holds and the integrity of name \( N_P \) is trusted then \( D_P \) can be trusted, too.
11.2.1 Descriptor Details

Descriptors for Programs and Data. Digital computers use bit strings to represent programs and data, whether that object is being stored by main memory or by a long-term storage device like a disk. To prevent attackers from counterfeiting descriptors for such objects, a descriptor $d_O$ for an object $O$ should be a collision and preimage resistant function of the bit string $B_O$ that is representing $O$. So an obvious choice for $d_O$ is a cryptographic hash $H(B_O)$.

This scheme for generating descriptors does mean that copies of an object $O$ incorporating pointers and located at different addresses will be given different descriptors. If that differentiation is undesirable then $O$ might be represented as a pair $(ARM_O, AID_O)$, where $ARM_O$ is an address-relocation map and $AID_O$ is an address-independent description. Hash $H(AID_O)$ is a descriptor for object $O$ that does not change for copies at different locations. Such a descriptor, however, would be insensitive to differences in $ARM_O$, and those differences might well cause copies of $O$ to satisfy different sets of properties.

Descriptors for Hardware Processors. If each hardware processor $hw$ has a unique name $N(hw)$ then using $H(N(hw))$ as the descriptor $d_{hw}$ for $hw$ prevents substitution of a different processor that will have the same descriptor. The problem of defining descriptors for hardware processors is now reduced to the problem of assigning names to hardware processors.

We might have each hardware processor provide a read-only register that contains a unique name for use in forming the processor’s descriptor. That approach, however, is easy to subvert. A program able to read this register could retrieve the processor’s name for use in an emulator implemented by software and executed on a different processor. That emulator would then be able to impersonate the original hardware processor, subverting the association of the name with the specific hardware processor.

We prevent such impersonations by deriving a name $N(hw)$ for a hardware processor $hw$ from a unique signing key $k_{hid}^{hw}$ that $hw$ stores, uses, but never reveals. Tamper-proof packaging prevents attackers who have physical access to $hw$ from learning or changing $k_{hid}^{hw}$. Key $k_{hid}^{hw}$ could be generated and installed by the manufacturer of $hw$ when the device is fabricated or the device itself could generate $k_{hid}^{hw}$ by, for example, leveraging fixed and unique timing differences inevitably present in a specific semiconductor chip.

A processor instruction to compute $k_{hid}^{hw}$-signed messages then allows corresponding verification key $K_{hid}^{hw}$ to serve as $N(hw)$. To make this hardware processor naming scheme concrete, code for checking whether $N(hw) = K$ holds is given in Figure 11.1.\(^5\) The routine assumes that a register $qkr_{id}$ comes preloaded with $k_{hid}^{hw}$. It also assumes this register cannot be accessed except by executing in-

---

\(^5\)A manufacturer should not keep records of $k_{hid}^{hw}$. Otherwise, attackers could learn $k_{hid}^{hw}$ by compromising the database being stored at the manufacturer.

\(^6\)K-S(m) used in Figure 11.1 evaluates to true if $m$ is digitally signed by a private key corresponding to verification key $K$. 

August 10, 2020  Copyright Fred B. Schneider  All rights reserved.
11.2. Naming Schemes for Measured Principals

\[ \text{AuthHW: function}(K, r) \]
\[ \quad \text{quote}(\text{qkr}_\text{id}, r, \text{out}) \]
\[ \quad \text{if } K \text{-S}?(\text{out}) \]
\[ \quad \quad \text{then return( true )} \]
\[ \quad \quad \text{else return( false )} \]
\[ \quad \text{end AuthHW} \]

Figure 11.1: Check if \( N(hw) = K \) by Using Nonce \( r \)

struction\(^7\) `quote(qkr\_id, r, out)`, which computes \( k\_id\_hw\text{-S}(r) \) and writes that value into memory \( out \). Therefore, if \( \text{AuthHW}(K, r) \) returns \text{true} then \( N(hw) = K \) holds and the underlying processor has name \( K \).

When needed, information about the instruction set architecture \( ISA_{hw} \) that processor with name \( K\_id\_hw \) implements would be conveyed using a certificate

\[ k_C\text{-S}(K\_id\_hw \text{ speaksfor } ISA_{hw}) \quad (11.3) \]

signed by the processor’s manufacturer or some other trusted party \( C \), where verification key \( K_C \) is a well known and \( ISA_{hw} \) is a subprincipal of \( C \), so that \( K\_id\_hw \text{ speaksfor } ISA_{hw} \) is implied by (11.3). This certificate need not be kept secret, so it might be stored in read-only processor memory and/or disseminated on demand by the processor’s manufacturer. The certificate is essential for defending against an attacker who picks a processor name \( K \) that corresponds to some private key \( k \) the attacker knows. If the attacker could choose \( K \) then the attacker could install \( k \) in an attacker-provided software emulator and use that to impersonate \( hw \). But an attacker would not be able to create an instance of certificate (11.3) for a public key \( K \) chosen by the attacker.

Firmware on some hardware processors can be upgraded, potentially changing \( ISA_{hw} \) for the device. Certificate (11.3) no longer conveys the full picture, because that certificate does not say what version of the firmware is installed. Firmware upgrades typically are performed as part of the processor’s boot or by a function that the existing firmware supports. In either case, prudence dictates that an upgrade request include a certificate that authenticates the specific changes and is signed by the processor’s manufacturer. The protocol that performs microcode updates would check this certificate before making changes and, after the changes are made, would store this certificate to document the current microcode version and to serve as a descriptor for the processor’s microcode.

Additional Names for Hardware Processors. If there is a unique name for each processor then descriptors and descriptions reveal whether two measured principals are sharing a processor. That, in turn, can reveal information about location, ownership, likely users, etc. We might desire that such information be hidden for privacy and other reasons—especially if the information is being included in messages sent to untrusted services.

\(^7\)See §11.3.3 for a detailed discussion of the `quote` instruction.
One solution is for a processor $hw$ to have one or more attestation identity keys, each a surrogate for $K_{id}^{hw}$ and usable as the name for $hw$. Validity of an attestation identity key $K_{A}^{hw}$ is conveyed by a certificate

$$k_T\cdot S(K_{A}^{hw} \text{ speaksfor } ISA_{hw})$$

that is signed by some trusted-third party $T$ having a well known verification key $K_T$. Certificate (11.4) would be generated by $T$ in response to a request that contains two certificates

$$k_C\cdot S(K_{id}^{hw} \text{ speaksfor } ISA_{hw}), \; k_{id}^{hw}\cdot S(K_{A}^{hw} \text{ speaksfor } K_{id}^{hw})$$

where the first certificate gives the instruction set architecture of $K_{id}^{hw}$ and the second gives the new key so, by transitivity of speaksfor, the same instruction set architecture is associated with attestation identity $K_{A}^{hw}$ as with original identity $K_{id}^{hw}$.

Descriptors for Properties. When using the descriptors discussed above for programs and data, an upgrade or other modification could lead to a different descriptor. Compare that with a descriptor that is derived from the statement of some property characterizing an artifact’s behaviors. Such a descriptor need not be changed if the artifact is modified in ways that do not affect the property. So descriptors for statements of properties are less brittle than descriptors computed directly from the artifacts themselves.

Descriptors for properties turn out to be essential. This is because descriptions, names of measured principals, and system configurations used to authorize gating functions do not have to be changed if applying a patch or upgrade leaves unchanged the properties of that resource. (The integrity of a patch would need to be verified before that patch is applied.) System configurations used to authorize gating functions also are easier to manage if different resource instances use the same descriptor.

The descriptor for a property would be derived from an attestation that binds some assertion to a source, where if the source is trusted then the assertion can be trusted. Various implementations are possible.

- A signed certificate from a trusted organization, where the contents identifies the property that is purported to hold. This property should be trusted if the certificate’s signer is trusted. A descriptor would be obtained by computing a hash function over the certificate representation.

- The output of an analyzer. The analyzer output can be trusted by $P$ provided (i) a descriptor for the code executed is equal to the descriptor for an analyzer that $P$ trusts and (ii) $P$ invoked the analyzer, provided the inputs, and received the outputs. A descriptor for the analyzer output is obtained by computing a hash function on the name of the measured principal that executes the analyzer, a descriptor for the analyzer code, and descriptors for the inputs.
11.2.2 Auxiliary Information

The identity or implementation details of the resources associated with individual descriptors could be important for deciding whether to trust a measured principal. So trust assessments for measured principals are facilitated when auxiliary information is made available for each descriptor.\(^8\)

**Requirements for Auxiliary Information.** Auxiliary information \(Aux_i\) associated with a descriptor \(d_i\) should allow:

(i) identification and retrieval of the objects involved in calculating descriptor \(d_i\),

(ii) recalculation of \(d_i\) from those objects, and

(iii) assessment of whether those objects should be trusted.

One example of such auxiliary information is the name of a software release (e.g., `Linux4.8.0-36-generic`) or the name of a file or directory on a public server. This information would allow relevant files to be downloaded and a descriptor \(d'_i\) independently computed for that downloaded content. If its descriptor \(d'_i\) satisfies \(d'_i = d_i\) then the files identified in \(Aux_i\) correspond to \(d_i\), so properties of the resources \(d_i\) describes can be validated by analyzing the downloaded files. Analysis of the downloaded files might involve testing, formal methods, or simply checking that the files are from some trusted software provider.

As another example, auxiliary information \(Aux_i\) for a digital certificate \(cert\) associated with a descriptor \(d_i\) might simply be \(cert\) itself. This auxiliary information is not only useful with descriptors for properties. It also is useful with descriptors for hardware processors, where certificate (11.3) would serve as the auxiliary information.

The integrity of auxiliary information is, of course, crucial. But additional mechanism is not needed for protecting that integrity, provided the integrity of measured principal names is being protected. The approach we employ is analogous to how integrity of descriptions is handled.

**Checking Integrity of Auxiliary Information.** To check the integrity of auxiliary information associated with a measured principal having a name \(N_P\)

1. Obtain and validate the integrity of a candidate description \(D_P\) for \(N_P\) by checking that \(N_P = \mathcal{N}(D_P)\) holds.

2. For each descriptor \(d_i\) in candidate description \(D_P\), check the integrity of associated auxiliary information \(Aux_i\) by using the above Requirements for Auxiliary Information.

   (a) Obtain copies of the associated resources, based on the information provided in requirement (i).

---

\(^8\)There is no widely accepted term for what we call auxiliary information. Some authors require that this information be incorporated into a descriptor; others call it a load list.
(b) Use those copies to compute a descriptor $d'_i$ (as per requirement (ii)) and compare $d'_i$ with $d_i$. If $d_i = d'_i$ holds then the integrity of Aux$_i$ has not been compromised.

Once integrity has been established for the auxiliary information then copies of the resources fetched in step 2a can be analyzed (as requirement (iii)) allows) to make a trust assessment for $N_P$.

11.3 Hardware Support for Gating Functions

Hardware support for gating functions typically extends a processor’s instruction set architecture by adding registers and system-mode instructions. To illustrate, we describe an idealized embodiment. It includes:

- Instructions to update measurement registers. All but one of the measurement registers is reset whenever the processor reboots. These registers are used for defining configuration constraints.

- Instructions to generate fresh cryptographic keys and store into key registers those keys that must be kept secret.\(^9\) Values stored in key registers persist across reboots. The key registers include sealing-key registers $\text{skr}_1$, ..., $\text{skr}_N$, quoting-key registers $\text{qkr}_{\text{id}}$, $\text{qkr}_1$, ..., $\text{qkr}_N$, and unbinding-key registers $\text{ukr}_1$, ..., $\text{ukr}_N$.

- Instructions that use a key $K$ in a specified key register and compute a gating function $[K-F](\cdot)$ to perform
  - $\text{sealing}$ to protect the confidentiality and integrity of local content,
  - $\text{quoting}$ to establish authenticity of locally produced content, or
  - $\text{binding}$ to import remote content but only if the local system satisfies certain configuration constraints,

where $\text{Config}([K-F])$ is defined in terms of measurement register contents.

In this design, confidentiality of cryptographic keys is protected because (i) unencrypted keys never leave key registers and (ii) instructions that use the contents of a key register are computing cryptographic functions that, by design, reveal nothing about the key. Moreover, even though values in key registers persist across reboots, those keys cannot be abused. This is because the values of the measurement registers do not persist across reboots. So for a gating function to use the contents of a key register after a reboot, the measurement registers would have to be returned to the values they had before the reboot. That implies the same set of measured principals would have to be running after the reboot as before the reboot (and presumably those measured principals are trusted to use the keys).

\(^9\)To avoid the cost of unnecessary mechanism, keys that need not be kept secret are stored in memory rather than being stored in key registers.
11.3. Measurement Registers and Constraints

Measurement registers $mr_0, mr_1, ..., mr_N$ are used for defining configuration constraints.

Measurement register $mr_0$ is automatically incremented each time the system reboots. By including this counter in a configuration constraint, we can create an ephemeral key—a key that becomes unusable after a reboot occurs. Ephemeral session keys defend against TOCTOU (time-of-check, time-of-use) attacks in which an attacker instigates a reboot to load and start malware that will run in place of principals that had been authenticated by some remote service, causing the remote service to confuse messages from the malware with messages being from the authenticated principals.

Measurement registers $mr_1, mr_2, ..., mr_N$ (unlike $mr_0$) are reset to 0 whenever the system reboots. System mode instructions $MRreset$ and $MRextend$ update these registers. In the description of $MRextend$ below, $mem$ denotes the contents (not the address) of some memory region. That contents often is the executable for a program being incorporated into the configuration, but $mem$ could be initialization data.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$MRreset(mr_i)$</td>
<td>$mr_i := 0$</td>
</tr>
<tr>
<td>$MRextend(mr_i, mem)$</td>
<td>$mr_i := H(mr_i \cdot H(mem))$</td>
</tr>
</tbody>
</table>

$MRreset$ and $MRextend$ are the only instructions that change $mr_1, mr_2, ..., mr_N$. Values stored by these registers thus reflect past and/or current contents of one or more memory regions. Notice that $MRextend$ calculates values according to (11.2), making $mr_1, mr_2, ..., mr_N$ ideally suited to storing names of measured principals.

*Configuration Constraints.* A configuration constraint $C$ is specified by listing a subset of the measurement registers and, for each, giving a value it must store. Set of pairs

$$C = \{ \ldots, (i, v_i), \ldots \}$$

specifies configuration constraint

$$\bigwedge_{(i, v_i) \in C} mr_i = v_i. \quad (11.5)$$

and can involve any subset of measurement registers $mr_0, mr_1, ..., mr_N$.

A configuration constraint $C$ is *satisfied* during execution if and only if the current values of the measurement registers that $C$ mentions agree with the values $C$ prescribes:

$$ConfigSat(C): (\forall i: (i, v_i) \in C \Rightarrow mr_i = v_i)$$

Hardware support for gating functions typically will associate a separate configuration constraint with each key register (rather than associating a configuration constraint with each gating function). This limitation is not significant when, as is typical, a single predetermined gating function (and perhaps its
inverse, for symmetric cryptography) provides the sole way to access the value in a given key register.

Systems often follow conventions that associate specific measurement registers with the different layers of the system software stack. For example, a convention might stipulate that \( \text{mr}_1 \) contain the name of a measured principal for the operating system, \( \text{mr}_2 \) the name of a measured principal for middleware, and so on.

To base trust on a configuration constraint, we must have reason to believe that principals invoke \( \text{MExtend} \) instructions as appropriate. That belief about principals can be justified if (i) they are measured principals, (ii) their names derive from descriptions for software we analyzed and are therefore prepared to trust, and (iii) we also trust the software that loaded those measured principals. So we have built a chain of trust by following execution back to the boot-loader firmware and ultimately the hardware processor. Trust in the chain depends on having trust in every link of the chain; trust that a link (except the first) is what it purports is derived from trust in the actions by the code in the preceding link; trust in the first link—called the root of trust for the chain—must be based on information from some external source.

### 11.3.2 Seal and Unseal

Instructions seal and unseal are gating functions to convert between unsealed bit strings and \( K/C \)-sealed bit strings, where \( K \) is a symmetric key stored in some sealing-key register, and \( C \) is a configuration constraint.

- Reading a \( K/C \)-sealed bit string reveals nothing about \( K \) or about the unsealed bit string from which it was derived.

- Updates to a \( K/C \)-sealed bit string are not blocked but cause subsequent execution of unseal to fail.

Because writes to \( K/C \)-sealed bit strings are not blocked, availability of a \( K/C \)-sealed bit string is compromised by writing to it but integrity is not compromised.

**SKRgen** generates a fresh symmetric key, loads that key into an indicated sealing-key register, and associates a configuration constraint defined by the current values of the measurement registers named in a set \( crSet \) stored in memory. The sealing key register comprises two fields: \( \text{skr} \_\text{key} \), which stores the key, and \( \text{skr} \_\text{config} \), which stores the configuration constraint.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
</table>
| \( \text{SKRgen}(\text{skr}_i, \text{crSet}) \) | \( \text{skr}_i \_\text{key} := \text{fresh symmetric key} \)  \\
|                           | \( \text{skr}_i \_\text{config} := \{(i, v_i) | i \in \text{crSet} \land \text{mr}_i = v_i\} \) |

Instruction seal is a gating function for shared-key authenticated encryption function \( K \cdot \text{E}^K() \); instruction unseal is a gating function for corresponding
decryption function $K \cdot D^h(\cdot)$, which fails rather than decrypting an argument not produced by $K \cdot E^h(\cdot)$.\footnote{The straightforward implementation for $K \cdot E^h(\cdot)$ and $K \cdot D^h(\cdot)$, known as encrypt then MAC, splits symmetric key $K$ into two equal size pieces: $K = K_1 \cdot K_2$. $K_2$ is used to generate a message authentication code, which becomes part of a pair that $K \cdot E^h(\cdot)$ returns: $K \cdot E^h(m) = (K_1 \cdot E(m), H(K_1 \cdot E(m) \cdot K_2))$. $K \cdot D^h(x, h)$ returns $K_2 \cdot D(x)$ only if $h = H(x \cdot K_2)$ holds; otherwise the invocation of $K \cdot D^h(x, h)$ fails. See Bellare and Namprempre [4] for strengths and weakness of alternative constructions.}

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>seal(skr, in, out)</td>
<td>$\text{out := skr}.key \cdot E^h(in)$</td>
</tr>
<tr>
<td>unseal(skr, in, out)</td>
<td>$\text{if ConfigSat(skr}.config$ then $\text{out := skr}.key \cdot D^h(in)$ else fail</td>
</tr>
</tbody>
</table>

Any program can invoke seal and encrypt information—indeed, independent of the system’s configuration. The resulting encrypted content can be recovered only by using unseal and only when the system satisfies the configuration constraint associated with the sealing-key register that seal used. And unseal fails if the encrypted value is altered (because execution of $K \cdot D^h(in)$ then fails).

By sealing the state that is being saved between executions of a given program or service, we protect confidentiality of that state from attackers—including across system reboots. Use of a restrictive configuration constraint can even protect sealed data from access by a version of the system that performed the sealing but subsequently was compromised. However, software upgrade then needs a protocol to migrate sealing-key registers from requiring the pre-upgrade configuration to requiring the configuration present after the upgrade is complete. Here is a sketch of such a protocol.

**Sealed-State Upgrade Protocol.**

1. System invokes unseal to transform all sealed bit strings data into unsealed bit strings.
2. Perform software upgrade. Upgraded system is now executing and has access to unsealed bit strings generated in step 1.
3. Upgraded system resets then reloads measurement registers and reprovisions the sealing-key registers (using configuration constraints that are defined using updated values in measurement registers).
4. Upgraded system, using reprovisioned sealing-key registers, invokes seal to transform unsealed bit strings from step 1 back into sealed bit strings.

In this protocol and elsewhere, indirection is often used to lower the cost of sealing and unsealing large amounts of data because, to reduce the costs for a rarely used feature, hardware implementations of seal and unseal often
are slow compared with running code for shared-key encryption and decryption directly on general purpose hardware. So we protect state by running a shared-key encryption routine on the processor, then use `seal` to protect that shared key. This two-level scheme, however, has risks. A cryptographic key is exposed in memory (rather than residing in a key register) while running the encryption and decryption programs on the general purpose hardware. And attackers have an easier time getting access to keys in memory than to keys that reside in key registers protected by configuration constraints.

**Key Archives.** Extensions of `seal` and `unseal` facilitate time multiplexing the key registers. `KRseal` seals the contents of some specified subset of the key registers (including the key and configuration constraint fields for each) and stores that in memory as a `key archive`. `KRunseal` restores those values to their original key registers. In the descriptions that follow, \( kr_j \) denotes an arbitrary key register, \( krSet \) is a subset of the key registers, and \( ska \) is the memory address for the key archive.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
</table>
| `KRseal(skr_i, krSet, ska)` | `let keyArchive = \{ (j, v_j) \mid j \in krSet \land kr_j = v_j \}`  
|             | `in ska := seal(skr_i, keyArchive)` |
| `KRunseal(skr_i, ska)` | `let keyArchive = unseal(skr_i, ska)`  
|             | `in for (j, val_j) \in keyArchive do kr_j := val_j` |

Because a key archive is a sealed bit string, reading a key archive reveals nothing about keys or configuration constraints it contains. In addition, writes to a key archive cannot alter the values of keys or configuration constraints that will be restored, because the invocation `unseal` by `KRunseal` will fail for a key archive that has been modified.

### 11.3.3 Quoting

A quoted bit string will have an unforgeable digital signature affixed. By using a sufficiently restrictive configuration constraint, a valid signature implies that the quoted bit string was generated on a specified processor while executing specified binaries that read certain files. That, in turn, can serve as a basis for deciding whether information conveyed by that quoted bit string should be trusted.

To generate quoted bit strings, we use gating functions based on cryptographic functions for generating and verifying digital signatures.

- Signing keys along with associated configuration constraints for a quoting key register `qkr` are stored in fields `qkr.key` and `qkr.config`.
  - Quoting-key register `qkr_id` has a fixed value and attests to no configuration constraints
    \[
    qkr_{id}.key = k_{hw}^{id} \quad qkr_{id}.config = \emptyset
    \]
11.3. Hardware Support for Gating Functions

**CheckSig**: function\((K, qIn, out)\)

\[
\begin{align*}
\text{if } &\text{-}\text{Sat}(qIn) \text{ then } \text{out} := K\cdot\text{Sat}(qIn) \\
\text{else } &\text{fail}
\end{align*}
\]

end CheckSig

Figure 11.2: Check Digital Signature on \(qIn\)

where \(k_{\text{id}}\) is a unique signing key associated with the processor \(hw\) that hosts this register.\(^{11}\)

- Quoting-key registers \(qkr_1, ..., qkr_Q\) can be reloaded.

- Verification keys for checking digital signatures and corresponding certificates to identify those verification keys are stored in memory, since that information is not secret.

An instruction \(\text{QKRgen}\) provisions a quoting-key register by (i) generating a fresh signing key, (ii) associating a specified configuration constraint, and (iii) depositing at the indicated memory address a \(k_{\text{id}}\)-signed certificate that gives the verification key for signatures generated using this fresh signing key.

<table>
<thead>
<tr>
<th>Instruction (\text{QKRgen}(qkr, crSet, mem))</th>
<th>Operation</th>
</tr>
</thead>
</table>
| \(qkr, \text{config} := \{(i, v_i) \mid i \in crSet \land mr_i = v_i\}\) | \(\text{let } k/K \text{ be a fresh private/public key pair}
\(\text{in } qkr, \text{key} := k\)
\(\text{mem} := qkr_{\text{id}}\cdot\text{key}\cdot\text{Sat}(qkr\text{ key} : iK)\) |

Instruction \(\text{quote}\) is a gating function based on a cryptographic function \(k\cdot\text{Sat}(\cdot)\) that generates digitally signed bit strings.

<table>
<thead>
<tr>
<th>Instruction (\text{quote}(qkr, in, qout))</th>
<th>Operation</th>
</tr>
</thead>
</table>
| \(\text{if ConfigSat(qkr.config)}\) | \(\text{then } qout := qkr, \text{key}\cdot\text{Sat}(\text{sig} : iin)\)
\(\text{else fail}\) |

Notice, bit strings that \(\text{quote}\) produces cannot be mistaken for certificates that \(\text{QKRgen}\) produces—quoted bit strings contain prefix “\(\text{sig} : i\)” but the certificates contain a different prefix, “\(qkr\text{ key} : i\)” As we shall see, each of the various instructions that generates a certificate will include a distinctive prefix, making it impossible to mistake what instruction was used to produce a given certificate. Attacks that repurpose certificates now are more difficult to devise.

Figure 11.2 gives code for a function \(\text{CheckSig}(K, qIn, out)\) that uses a verification key \(K\) to check the signature on \(qIn\). If the signature is valid, then the bit string that was signed to produce \(qIn\) is stored in \(out\).

To trust results produced by executing a gating function \([K\cdot\text{F}](\cdot)\) requires knowledge of the configuration constraint associated with the key register that

\(^{11}\)Recall (see page 308), corresponding verification key \(K_{\text{id}}\) will be well known.
stores \( K \). Instruction \( \text{KRgetConf} \) provides that information; it stores into memory a \( \text{id}_{\text{hw}} \)-signed certificate for the configuration constraint currently associated with a specified key register. A second instruction \( \text{KRgetCurConf} \) reveals the current configuration, so a program can test whether a key register’s configuration constraint is currently satisfied; \( \text{KRgetCurConf} \) stores into memory a \( \text{id}_{\text{hw}} \)-signed certificate giving the current values for any designated set of measurement registers. Argument \( r \) allows a nonce to be incorporated into certificates produced by these instructions, which facilitates defending against replay attacks.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{KRgetConf}(kr, r, out) )</td>
<td>( out := \text{qkr}_\text{id}.\text{key-S}(\text{keyConfig}: \text{irr} \text{kr}, \text{config}) )</td>
</tr>
<tr>
<td>( \text{KRgetCurConf}(\text{crSet}, r, out) )</td>
<td>( cc := { (i, v_i) \mid i \in \text{crSet} \land \text{mr}<em>i = v_i } ) ( out := \text{qkr}</em>\text{id}.\text{key-S}(\text{curConfig}: \text{r} \text{cc}) )</td>
</tr>
</tbody>
</table>

Certificates from \( \text{KRgetCurConf} \) that include \( \text{mr}_0 \) in \( \text{crSet} \) are useful for defending against attacks that replay outdated immutable data objects, including old descriptions of measured principals and old configuration constraints. For that defense, the certificate is included as part of the object so that the certificate contents affects the descriptor for that object and, consequently, it affects the name of any measured principal that incorporates the object. That means a measured principal using an outdated version of the data object will be assigned a different name and have different privileges than a measured principal that uses a current version of the data object.

### 11.3.4 Binding and Unbinding

If decryption with a private key \( k \) is a gating function \( [k-D](\cdot) \) then information that is encrypted under the corresponding public key \( K \) can be recovered only by systems satisfying configuration constraint \( \text{Config}([k-D]) \). By encrypting some information under public key \( K \), we are binding that information to a specific system—the system that is able to use gating function \( [k-D](\cdot) \) to invert that encryption by having a key register containing private key \( k \) and satisfying configuration constraint \( \text{Config}([k-D]) \). Gating function \( [k-D](\cdot) \) thus implements the corresponding unbinding.

Encryption under a public key can be performed by a remote processor, because public keys can be shared freely. That location flexibility distinguishes binding from sealing. Sealing must be performed on the same processor as the unsealing, since both operations must access the same key register. By allowing encryption and decryption to be performed on different processors, binding and unbinding provide a way for a remote client to submit an encrypted request to some service and have assurance that the request can be decrypted only by specific code running in its intended system configuration.

Hardware support for binding and unbinding includes unbinding-key registers to hold private keys, an instruction \( \text{UKRgen} \) to generate a public/private key pairs with appropriate certificates, and an instruction \( \text{UKRdec} \) that implements a gating function for decryption using a private key stored in some unbinding-key
11.4 Software Extensions

UKRgen stores into memory a \( k_{id} \)-signed certificate that identifies an unbinding-key register \( ukr_i \) and gives a public key \( K \) for binding content to configurations satisfying \( ukr_i.config \). To bind content, it suffices to encrypt using \( K \); key registers (and constraints) play no role in that computation.

### 11.4 Software Extensions

In the hardware support outlined above for gating functions, all clients share a single set of measurement registers and key registers. If, instead, each client had exclusive access to separate sets of these registers then clients would not have to cooperate or trust each other. Such functionality can be provided for user-mode clients by a system-mode software layer that implements a run-time environment.

The new run-time environment will provide software emulations for instructions that use measurement registers and key registers, with separate instances of these registers emulated for each client. Recall from §11.3, instructions that access these registers are system mode. The standard recipe for giving system-mode instructions new semantics then is applicable: Execute clients in user mode, so client execution of a system-mode instruction from §11.3 causes a trap that transfers control to an instruction emulator running in system-mode.

**GFmult** (detailed below) is such an emulator. It uses a table \( R \) to store a separate set of configuration register values and key register values for each different client.

- \( R[c].mr_i \) contains the value of client \( c \)'s configuration register \( mr_i \),
- \( R[c].kr_i.config \) contains the configuration constraints associated with client \( c \)'s key register \( kr_i \), and
- \( R[c].keyAr \) contains a key archive for client \( c \)'s key registers.

Code to emulate the instructions of §11.3 is given in Figure 11.3. It assumes variable \( c \) identifies the client that executed the instruction to be emulated.

The tricky part in the design of **GFmult** was overcoming the absence of an instruction to restore the values in measurement registers when switching from one client to another. That led to a **GFmult** emulation of \( \text{MRreset}, \text{MRextend}, \)
<table>
<thead>
<tr>
<th>System-mode Instruction</th>
<th>$GF_{mult}$ Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRreset($mr_i$)</td>
<td>if $i \neq 0$ then $R[c].mr_i := 0$ else fail</td>
</tr>
<tr>
<td>MRextend($mr_i, mem$)</td>
<td>if $i \neq 0$ then $R[c].mr_i := \mathbb{H}(R[c].mr_i \cdot \mathbb{H}(mem))$ else fail</td>
</tr>
<tr>
<td>SKRgen($skr_i, crSet$)</td>
<td>if $i = 1$ then fail $R[c].skr_i.config := { (i, v_i)</td>
</tr>
<tr>
<td>seal($skr_i, in, out$)</td>
<td>if $\text{CliConfigSat}(R[c].skr_i.config) \land i \neq 1$ then seal($skr_i, in, out$) else fail</td>
</tr>
<tr>
<td>unseal($skr_i, in, out$)</td>
<td>if $\text{CliConfigSat}(R[c].skr_i.config) \land i \neq 1$ then unseal($skr_i, in, out$) else fail</td>
</tr>
<tr>
<td>KRseal($skr_i, krSet, ska$)</td>
<td>if $i \neq 1$ then KRseal($skr_i, krSet, ska$) else fail</td>
</tr>
<tr>
<td>KRunseal($skr_i, ska$)</td>
<td>if $i \neq 1$ then KRunseal($skr_i, ska$) else fail</td>
</tr>
<tr>
<td>QKRgen($qkr_i, crSet, mem$)</td>
<td>$R[c].qkr_i.config := { (i, v_i)</td>
</tr>
<tr>
<td>quote($qkr_i, in, qOut$)</td>
<td>if $\text{CliConfigSat}(R[c].qkr_i.config)$ then quote($qkr_i, in, qOut$) else fail</td>
</tr>
<tr>
<td>KRgetConf($kr_i, r, out$)</td>
<td>info := keyConfig ${ \text{id}</td>
</tr>
<tr>
<td>KRgetCurConf($crSet, r, out$)</td>
<td>cc := ${ (i, v_i)</td>
</tr>
<tr>
<td>UKRgen($ukr_i, crSet, mem$)</td>
<td>$R[c].ukr_i.config := { (i, v_i)</td>
</tr>
<tr>
<td>UKRdec($ukr_i, in, out$)</td>
<td>if $\text{CliConfigSat}(R[c].ukr_i.config)$ then UKRdec($ukr_i, in, out$) else fail</td>
</tr>
</tbody>
</table>

Figure 11.3: $GF_{mult}$ Implementation Details
and KRgetCurConf that uses $R[c].mr_i$ in place of hardware register $mr_i$. Such an emulation requires that (i) $R[c].mr_0$ be initialized to the value in hardware register $mr_0$ when $GFmult$ starts and (ii) emulations for instructions (e.g., seal, unseal, quote, UKRdec) use KlConfigSat($c$, kr$_i$): $(\forall j: (j, v_j) \in R[c].kr_i.config \Rightarrow R[c].mr_j = v_j)$ for testing whether configuration constraints for a client $c$’s key register $kr_i$ are satisfied by values in client $c$’s measurement registers.

$GFmult$ instruction emulations are implemented in terms of the hardware instructions from §11.3 that reference key registers. For the invocations of those instructions to succeed, associated configuration constraints must be satisfied. In the emulations of instructions SKRgen, QKRgen, and UKRgen for provisioning a key register, set $GFset$ specifies a configuration constraint that will be satisfied by values expected to be in the hardware measurement registers whenever $GFmult$ is being executed. And the $GFmult$ emulation for an instruction that accesses a client’s version of key register $kr_i$ checks the configuration constraint specified by set $crSet$ that the client supplies when provisioning $kr_i$—for client $c$, that configuration constraint is stored in $R[c].kr_i.config$ and checked against values in $R[c].mr_i$ for $i \in crSet$.

The hardware key registers—just like other processor registers being multiplexed across clients—must contain appropriately restored values when execution of a client is resumed. To achieve that:

- Interrupt handlers use KRseal to save in $R[c].keyAr$ a key archive containing values currently being stored by the hardware key registers whenever a client $c$ is interrupted. Although all of these registers have $GFset$ for their configurations (so that $GFmult$ emulations are able to use the registers), the cryptographic key value stored by the register is one associated with the client.

- The scheduler that selects a next client for execution (i) sets the value of $c$ accordingly, and (ii) invokes KRunseal with key archive $R[c].keyAr$ to restore the values of the key registers for client $c$.

A sealing key register is required for creating the key archives employed above for multiplexing the key registers. $GFmult$ uses skr$_1$ for this purpose; other client access to skr$_1$ is prevented by having the emulations for SKRgen, seal, unseal, KRseal, KRunseal fail when skr$_1$ is an argument.

Two final points are worth mentioning about the code in Figure 11.3. First, the emulation of each instruction assumes key registers contain the same values as they had when the client invoked the instruction about to be emulated. An interrupt handler that transfers control to $GFmult$ thus must not alter key registers. Second, the same client should be resumed once an instruction emulation

\[^{12}\text{KRgetCurConf can be executed by } GFmult \text{ to obtain a certificate that includes the current value of } mr_0.\]
1. $R \rightarrow S: \langle r, P \rangle$, where $r$ is a fresh nonce.

2. $S$: Generate fresh public/private keys $K^\text{att}_P/k^\text{att}_P$ for use with a gating function $[k^\text{att}_P-S](\cdot)$, where $\text{Config}([k^\text{att}_P-S]) = \{ P \}$.

3. $S \rightarrow R: [k^\text{att}_S-S](r, P, K^\text{att}_P)$

4. $R$: Accept $K^\text{att}_P$ as a remote attestation key for $P$ provided:
   (a) Nonce $r$ and name $P$ received step 3 are the same values as sent in step 1.
   (b) $K^\text{att}_S$ verifies digital signature received step 3.

Figure 11.4: Remote Attestation by $R$ of $P$ Running on $S$

 completes. If, instead, another client is to be next executed then the key registers first must be saved in $R[c].keyAr$—either $GFmult$ or the scheduler could do this.

### 11.5 Remote Attestation

A remote attestation protocol returns to its initiator

- the name $P$ for a measured principal being executed by a remote host, and

- an attestation public key $K^\text{att}_P$ for verifying signatures on messages digitally signed by $P$.

To decide whether $P$ and messages that $K^\text{att}_P$ verifies can be trusted, it then suffices to obtain a candidate description $D_P$ along with associated auxiliary information, validate the integrity of that by Checking Integrity of Auxiliary Information (page 311), and then perform the necessary analysis to determine that the system $D_P$ describes satisfies expected properties, including that digital signatures verified by $K^\text{att}_P$ can be created only by executing $P$.

Note, if rebooting the processor that is running $P$ does not delete signing key $k^\text{att}_P$ then clients need to defend against TOCTOU attacks involving a reboot that is instigated between the remote attestation that $K^\text{att}_P$ provided and the subsequent sending of messages that $K^\text{att}_P$ verifies. One such defense is to include as a descriptor in $D_P$ a value that is incremented with each reboot. The value of $mr_0$ would work; but a certificate giving the current time and signed by a trusted source also would work.

### 11.5.1 A Remote Attestation Protocol

A remote attestation protocol is sketched in Figure 11.4. By communicating with some measured principal $S$, initiator $R$ learns name $P$ and attestation
11.5. Remote Attestation

public key $K^\text{att}_P$ for a measured principal that $S$ is executing. The protocol depends on the following assumptions.

(i) $R$ trusts $S$ and has an attestation public key $K^\text{att}_S$ that verifies signatures on messages digitally signed by $S$.

(ii) $S$ is the environment that executes $P$. Thus, description $D_S$ is a prefix of $D_P$ and $P = \mathcal{N}(D_P)$ holds.

(iii) $S$ implements gating function $[k^\text{att}_S(T)](\cdot)$, for generating digital signatures, as required in step 2 of the protocol in Figure 11.4.

Discharging (i) is straightforward when $S$ comprises a hardware processor $hw$ and the lowest levels of its software stack. Initiator $R$ provides a fresh challenge $r$ that $S$ uses when invoking instruction $K\text{RgetConf}$ to obtain a $k^\text{id}_{hw}$-signed certificate that attests $\{\mathcal{N}(D_S)\} = \text{Config}([k^\text{att}_S(T)])$ is the configuration constraint associated with a gating function $[k^\text{att}_S(T)](\cdot)$. This certificate is returned to $R$. Knowledge of verification key $K^\text{id}_{hw}$ and challenge $r$ allows $R$ to validate the source and timeliness of the certificate; knowledge of $D_S$ allows $R$ to compute $\mathcal{N}(D_S)$ and verify that this value defines the configuration constraint that the $k^\text{id}_{hw}$-signed certificate specifies. $R$ then concludes $K^\text{att}_S$ speaks for $S$. To discharge (ii) is simple, given some means to obtain description $D_P$. Finally, a manufacturer's certificate like (11.3) that gives the instruction set architecture for $hw$ provides a basis, in conjunction with $D_S$, for discharging (iii).

*Formal Analysis of Remote Attestation Protocol. Formalized in CAL, the goal of the protocol in Figure 11.4 is for initiator $R$ to obtain values for $P$ and $K^\text{att}_P$ satisfying

$$K^\text{att}_P \text{ speaks for } P.$$ (11.6)

This can be formally derived from the protocol and assumptions as follows.

Step 3 of the protocol in Figure 11.4, is formalized as

$$K^\text{att}_S \text{ says } (S.r \text{ says } (K^\text{att}_P \text{ speaks for } P))$$ (11.7)

where a subprincipal $S.r$ is being used to identify the activity that principal $S$ undertakes in processing a request accompanied by nonce $r$, so we have

$$S.r \text{ speaks for } S.$$(11.8)

Assumption (i) above implies

$$K^\text{att}_S \text{ speaks for } S.$$ (11.9)

We are using the following CAL inference rule, which formalizes how gating functions restrict the creation of digitally signed messages:

$$\text{GATING: } \{T\} = \text{Config}([k^\text{att}_T(T)](\cdot)) \implies \quad K^\text{att}_T \text{ speaks for } T.$$ (11.10)
Chapter 11. Measured Principals and Gating Functions

The hypothesis \( \{ T \} = \text{Config}(\{ k_{\text{att}} \}) \) is discharged by obtaining a \( t_{\text{att}} \)-signed certificate from \( S \), as discussed above.

Now, by using CAL inference rule (9.16) with (11.9) and with (11.7) we get

\[
S \text{ says (} S.r \text{ says (} K_{P}^{\text{att}} \text{ speaksfor } P \text{))}.
\]

Another use of CAL inference rule (9.16) but with (11.8) yields:

\[
S.r \text{ says (} S.r \text{ says (} K_{P}^{\text{att}} \text{ speaksfor } P \text{))} \tag{11.11}
\]

CAL inference rule \text{SAYS-E} with (11.11) infers:

\[
S.r \text{ says (} K_{P}^{\text{att}} \text{ speaksfor } P \text{)}
\]

Because \( S \) provides the execution environment for \( P \) (due to assumption (ii) above), we conclude that \( P \) is subprincipal of \( S \). So from CAL inference rule \text{SUBPRIN} we conclude \( S \text{ speaksfor } P \), and by transitivity of \text{speaksfor} CAL inference rule \text{DELEG-TRANS} with (11.8) we get \( S.r \text{ speaksfor } P \). Applying (9.16) we get:

\[
P \text{ says (} K_{P}^{\text{att}} \text{ speaksfor } P \text{)}
\]

An application of CAL inference rule \text{HAND-OFF} then yields (11.6), as desired.

11.5.2 Remote Attestation for System Startup

The remote attestation protocol just given depends on hardware support for gating functions. We now discuss a remote attestation protocol that does not require such hardware support. But unlike the protocol above, this second protocol requires that system startup undertake advance preparation for servicing future attestation requests.

System startup typically comprises \textit{stages}. These stages execute one at a time, starting with processor firmware and continuing to layers of system software. The exact sequence of stages will depend on the system configuration, including which co-processors and/or specific input/output devices are present. Each stage initializes its state, does some computation, determines its successor stage, loads code and data for that successor, and terminates by transferring control to its successor. Once a stage has terminated, it no longer participates in remote attestation protocols (or any other protocol, for that matter).

System startup that culminates in running a measured principal \( S \) produces a series \( D_{0}^{S}, D_{1}^{S}, \ldots, D_{n}^{S} \) of descriptions. Each \( D_{i}^{S} \) is the description for a stage and corresponds to a measured principal \( \mathcal{N}(D_{i}^{S}) \), with \( D_{n}^{S} \) satisfying \( \mathcal{N}(D_{n}^{S}) = S \). A \textit{system startup remote attestation protocol}

- associates a pair of public/private attestation keys \( K_{i}^{\text{att}}/k_{i}^{\text{att}} \) with each stage \( \mathcal{N}(D_{i}^{S}) \),
- restricts uses of \( k_{i}^{\text{att}} \) according to \( \text{Config}(\{ k_{i}^{\text{att}} \}) = \{ \mathcal{N}(D_{i}^{S}) \} \), and
- provides a set \( \text{AttCert}_{S} \) of certificates from which \( K_{i}^{\text{att}} \text{ speaksfor } \mathcal{N}(D_{i}^{S}) \) can be inferred.
11.5. Remote Attestation

1. $k^\text{att}_0 := k^\text{id}_\text{hw}; \quad K^\text{att}_0 := K^\text{id}_\text{hw}; \quad \mathcal{N}(D^0_S) := \mathcal{N}(\text{hw});$

2. for $i := 0$ to $n - 1$ do
   
   (a) $\mathcal{N}(D^i_S)$ loads software for its successor stage, creating description $D^{i+1}_S$. Compute name $\mathcal{N}(D^{i+1}_S)$.

   (b) $\mathcal{N}(D^i_S)$ obtains public/private attestation keys $K^\text{att}_{i+1}/k^\text{att}_{i+1}$ and installs a gating function $[k^\text{att}_{i+1}/i](\cdot)$, with $\text{Config}([k^\text{att}_{i+1}/i]) = \mathcal{N}(D^{i+1}_S)$.

   (c) $\text{AttCerts}_S := \text{AttCerts}_S \cup \{k^\text{att}_{i+1}/i((K^\text{att}_{i+1}, \mathcal{N}(D^{i+1}_S)))\}$

   (d) $\mathcal{N}(D^i_S)$ relinquishes control to $\mathcal{N}(D^{i+1}_S)$ if $\text{AuthStages}(\mathcal{N}(D^{i+1}_S))$ holds; otherwise $\mathcal{N}(D^i_S)$ halts.

Figure 11.5: System Startup Attestation Protocol

One such protocol is given in Figure 11.5. The resulting set of certificates $\text{AttCerts}_S$ can be used to construct a chain for justifying trust in measured principal $\mathcal{N}(D^n_S)$. In step 2a of the protocol, a stage $\mathcal{N}(D^i_S)$ loads its successor $\mathcal{N}(D^{i+1}_S)$. For step 2b, two approaches could ensure that private attestation key $k^\text{att}_{i+1}$ has not previously been revealed: (i) generate a fresh $k^\text{att}_{i+1}$, or (ii) use key registers and key archives to store $k^\text{att}_{i+1}$. In step 2d, $\text{AuthStages}(\mathcal{N}(D^{i+1}_S))$ implements checks to establish that next stage $\mathcal{N}(D^{i+1}_S)$ can be trusted and, thus, should next receive control.

A stage $\mathcal{N}(D^{i+1}_S)$ can be trusted if there is reason to believe it will execute the protocol and, therefore, will relinquish control only to a stage that itself satisfies this requirement for being trusted. That check is being abstracted here by a predicate $\text{AuthStages}$, which is assumed to be satisfied only by names of stages that will correctly execute the protocol. An implementation of $\text{AuthStages}$ might simply compare against a list of names that have already been analyzed and stored in read-only firmware memory. Or it could use a description and auxiliary information for a stage to determine whether that stage instantiates some predefined template, where code for steps 2a through 2d is already given in the template and where only certain state may be referenced by other code the stage executes.

Gating functions, key registers, and key archives are not the only way to ensure that an attestation private key $k^\text{att}_{i+1}$ is not revealed or abused. For executions of a system startup attestation protocol, deletion of keys can achieve the same effect if keys are stored in processor memory and a program running on the processor computes $k^\text{att}_{i+1} \cdot S(\cdot)$. Once stage $\mathcal{N}(D^{i+1}_S)$ starts executing, no predecessor will execute. So access restrictions that a gating function $[k^\text{att}_{i+1}/i](\cdot)$ would impose can be achieved simply by deleting $k^\text{att}_{i+1}$ when $\mathcal{N}(D^{i+1}_S)$ relinquishes control to its next stage.

A proof below shows that the certificates in $\text{AttCerts}_S$ suffice, as required, to establish $K^\text{att}_i \text{ speaks for } \mathcal{N}(D^i_S)$ for $0 \leq i \leq n$. That proof assumes certificates in $\text{AttCerts}_S$ are current rather than generated prior to the last reboot. The
1. $R \rightarrow S$: \texttt{Att: r} for a fresh nonce.

2. $S \rightarrow R$: $k_{hw}^{id} \cdot S(\texttt{AttReply: AttCert}_{SS}, D_S, v)$, where $v = k_{hw}^{id} \cdot S(r, m_r_0)$

3. $R$: Accept remote attestation certificates in $\texttt{AttCert}_{SS}$ provided:
   a. $K_{hw}^{id}$ verifies digital signatures received step 2.
   b. Nonce $r$ received step 2 is same value as sent in step 1.
   c. Using $\texttt{AttCert}_{SS}$, $D_S$, and $v$ received step 2 check that:
      • $\texttt{AttCert}_{SS}$ implies $K$ speaks for $S$ for some $K$.
      • $D_S$ satisfies $S = N(D_S)$,
      • $D_S$ incorporates a descriptor giving a value for $m_r_0$ that is consistent with $v$.

---

*Formal Analysis of System Startup Attestation Protocol.* We prove that $K_{att}^{id}$ speaks for $N(D^i_S)$ for $0 \leq i \leq n$ can be inferred from $\texttt{AttCert}_{SS}$ after a system having description $D_S$ has run System Startup Attestation Protocol of Figure 11.5. The proof is by induction on $i$.

---

Instruction $K_{getCurConf}$ described on page 318 provides the required functionality.
The base case is to prove \( K_0^{\text{att}} \text{ speaksfor } N(D_S^0) \). Step 1 of System Startup Attestation Protocol in Figure 11.5 sets \( N(D_S^0) \) to \( N(hw) \). By definition, \( N(hw) = K_{hw}^{\text{att}} \) holds. Since \( K_0^{\text{att}} \text{ speaksfor } K_0^{\text{att}} \) is trivially valid, substitution of equals for equals yields \( K_0^{\text{att}} \text{ speaksfor } N(D_S^0) \), as needed.

For the induction case, assume \( K_j^{\text{att}} \text{ speaksfor } N(D_S^j) \) for \( 0 \leq j \leq i \) can be inferred from the certificates in \( \text{AttCerts}_S \); we must show how to infer \( K_{i+1}^{\text{att}} \text{ speaksfor } N(D_S^{i+1}) \). From step 2c of System Startup Attestation Protocol, \( \text{AttCerts}_S \) includes a certificate that implies:

\[
K_i^{\text{att}} \text{ says } K_{i+1}^{\text{att}} \text{ speaksfor } N(D_S^{i+1}).
\]

The induction hypothesis implies \( K_i^{\text{att}} \text{ speaksfor } N(D_S^i) \), so from CAL inference rule (9.16), we conclude

\[
N(D_S^i) \text{ says } K_{i+1}^{\text{att}} \text{ speaksfor } N(D_S^{i+1}).
\]

Because each stage \( D_S^{i+1} \) is executing in an environment created by its predecessor \( D_S^i \), we have that \( N(D_S^{i+1}) \) is a subprincipal of \( N(D_S^i) \). CAL inference rule SUBPRIN thus implies

\[
N(D_S^i) \text{ speaksfor } N(D_S^{i+1})
\]

so CAL inference rule (9.16) derives

\[
N(D_S^{i+1}) \text{ says } K_{i+1}^{\text{att}} \text{ speaksfor } N(D_S^{i+1}).
\]

HAND-OFF from Figure 9.4 yields goal \( K_{i+1}^{\text{att}} \text{ speaksfor } N(D_{i+1}) \).

**Trusted Boot and Secure Boot.** In addition to their role in the remote attestation protocol of Figure 11.5, \( \text{AttCerts}_S \) and \( \text{AuthStages} \) are useful for establishing trust in the execution environment—here, measured principal \( S \)—that system startup creates. It suffices that description \( D_S \) start with the descriptor for \( hw \), followed by descriptors for everything executed since \( hw \) last rebooted: boot firmware, other firmware, IPL routines, kernel initialization, etc.

With a processor that supports trusted boot, software establishes trust in its environment by checking whether \( \text{AttCerts}_S \) contains what is expected. Usually, the processor will have a register \( rt \) (say) for maintaining some well known Collision Resistant and Preimage Resistant function of \( \text{AttCerts}_S \). Register \( rt \) is reset automatically on reboot and there are instructions (i) to read its current value and (ii) to recompute its value when an element is being added to \( \text{AttCerts}_S \).\(^{14}\) Provided a system startup stage that updates \( \text{AttCerts}_S \) (step 2c of Figure 11.5) also updates \( rt \), then reading \( rt \) suffices for checking whether \( \text{AttCerts}_S \) is consistent with some predetermined set of certificates characterizing an execution environment that can be trusted. As with \( N(D_S) \), changes to \( D_S \) have unpredictable effects on the value in \( rt \). That means installing a new

\(^{14}\)Thus, a measurement register (page 313) could serve as \( rt \).
input/output device, adding a co-processor, or updating the operating system must also update expectations for the value to be found in \( rt \).

With a processor that supports secure boot, predicate \( AuthStages \) is strengthened so that it is satisfied only by a certain sequence of stages where, as before, all of those stages implement the protocol of Figure 11.5. Step 2d of that protocol implies system startup will halt execution rather than transfer control to an unanticipated stage, thereby taking the view that unanticipated stages are attacks. Hardware support for secure boot involves (i) register \( rt \) discussed above (ii) processor read-only memory initialized with the allowed sequence of values that \( rt \) may hold during system startup, and (iii) an instruction to update \( rt \) that will halt the processor when the new value of \( rt \) does not agree with the next value in the sequence being stored in the read-only memory.

11.6 Uses and Abuses

Measured principals and gating functions enable systems to enforce security policies that cannot be circumvented by a computer’s owners or its operators.\(^{15}\) Such policies can allay fears about compromise to confidentiality or integrity for data being processed on client-controlled machines or in public clouds. Given those security guarantees, digital delivery of content enables new business models. Public clouds also become attractive computing utilities.

11.6.1 Some of the Uses

Disk Encryption. A laptop that is lost or stolen might fall into the hands of an adversary. We protect the confidentiality of information its disks store if disk blocks are encrypted so, even with the laptop in hand, adversaries cannot access the key.

Gating functions for sealing might seem like an obvious way to implement such disk encryption. But processor-provided gating functions often are slow, making them ill-suited for encrypting and decrypting inputs the size of a disk block. Therefore, shared-key cryptographic routines implemented by software are the better choice. Length-preserving shared-key encryption enables the capacity of an encrypted disk block to be the same as a plaintext disk block, so neither block addresses on the disk nor the disk layout itself need be changed to accommodate storing encrypted disk blocks in place of plaintext disk blocks. However, with no space available on the disk to incorporate redundant information, length-preserving encryption schemes cannot protect integrity. So disk encryption schemes typically protect the confidentiality of disk blocks but not their integrity.

To implement disk encryption, each laptop is provisioned with a unique, secret disk key; that key is generated when the laptop is booted for the first

\(^{15}\)Of course, the mechanisms used to support measured principals and gating functions must be trustworthy, which is likely to depend on assumptions about (i) physical access that attackers have to the hardware and/or (ii) the effectiveness of tamper-proof packaging.
11.6. Uses and Abuses

A disk key must reside in main memory (rather than residing in a key register) in order to be read by encryption routines implemented in software. So confidentiality of the disk key is protected (only) by memory isolation that an operating system and/or hypervisor enforces.

Some laptops offer a hibernation mode as an alternative to choosing between the power drain of normal operation versus the time delays of system startup after a shutdown. A naive design for entering hibernation mode would store to disk a plaintext image of main memory, thereby allowing the laptop to resume operation without incurring delays associated with decryption. But that exhibits a vulnerability, because attackers who obtain a device can learn the disk key by reading the memory image from disk. Therefore, the memory image for hibernation mode must be stored in encrypted form. To exit hibernation mode, that memory image must be decrypted—but that decryption should be undertaken only after the system authenticates the user.

Because main memory contents is assumed to be obliterated\(^\text{16}\) when a laptop is powered down, the disk key must be stored by some non-volatile device. To protect this stored copy of the disk key, it is sealed using a hardware-implemented gating function. The associated sealing-key register has a configuration constraint that authorizes only a single measured principal, which comprises the laptop’s processor, non-volatile storage devices, boot sequence, and system software. That configuration constraint ensures an adversary cannot retrieve the disk key by replacing the processor, by connecting the non-volatile devices to a different computer, or by running different system software.

Having the disk key reside memory (rather than in a key register) also facilitates recovery of disk contents if the laptop fails. Encrypted disk blocks might be retrieved from a backup copy stored elsewhere, retrieved by connecting the disk to a different processor (perhaps because the original processor has failed), or retrieved by using the laptop after updates to the operating system (perhaps distributed with malicious intent) prevent access to the processor’s sealing key register because the name of the measured principal associated with the operating system has changed. Therefore, recovery of data from the encrypted disk blocks is made feasible if, whenever a new disk key is generated

- the new disk key is copied to removable media that is kept someplace physically secure and/or
- the new disk key, encrypted using a recovery key (perhaps just a long passphrase), is stored someplace likely to remain available, and the recovery key is kept secret.

By having routines to encrypt and decrypt disk blocks be part of the operating system, we avoid the need to include this functionality in each application

---

\(^{16}\)How long data in a volatile memory can still be read after power is removed depends on the semiconductor device technology. For some device technologies, this window can be extended by an attacker who perpetrates a so-called cold boot attack. First, the attacker chills the memory chips by spraying liquid nitrogen or compressed air. Then, once the memory chips have been chilled, the attacker reboots the system and executes a program that performs a memory dump onto an I/O device that the attacker later connects to a different system.
that uses the disk. A design that assumed encrypt and decrypt routines were inside the disk driver might seem clean, but it would require cooperation from disk manufacturers, since they provide the device drivers. Therefore, the most practical design is to leave disk drivers unchanged and have the operating system provide separate caches for encrypted and for unencrypted disk blocks. The disk driver would access the cache of encrypted disk blocks; input/output operations called by applications would access the cache of plaintext disk blocks; and operating system routines would perform cryptographic operations, as needed, to transform and move blocks between the encrypted and plaintext block caches. Note that boot blocks on the disk are not be encrypted, since those blocks are read from disk and executed before the operating system is available to perform decryption.

Cloud-Hosted Services. Gating functions enable cloud-hosted servers to resist attacks intended to change server code or to compromise confidentiality and/or integrity of server state stored outside of main memory. The server is implemented as a measured principal and provisioned with a sealing key, a quoting key, and an unbinding key. The sealing key is used to protect the confidentiality of information the server stores; the quoting key allows responses from the server to be authenticated by receivers; and the unbinding key allows clients to send confidential messages to the server.\(^\text{17}\)

However, the hosting environment for the server also must provide certain functionality.

- **Memory isolation.** Various approaches can be employed to isolate server state in main memory from other execution in the hosting environment. The server might be allocated its own processor, its own virtual machine, or leverage memory isolation an operating system provides for its processes. Caches and other shared processor resources, though, can become covert channels that compromise confidentiality.

- **Measured principals and gating functions.** This functionality could be provided by hardware, by software, or by some combination. A separate and independent sealing key, unbinding key, and quoting key is required for each server. Keys provisioned for a measured principal \(P\) that is implementing a server must be restricted so that only \(P\) can use them.

Clients of a cloud-based server must trust that server and its hosting environment. For a server implemented as a measured principal, its description \(D_P\) would identify what hardware and software components must be trusted. So a client can get assurance by obtaining a candidate for description \(D_P\) and checking whether it is authentic: The client first uses a remote attestation protocol to learn the name \(P\) (say) for the measured principal that implements the server and checks whether \(N(D_P) = P\) holds. Then the client retrieves the auxiliary information associated with description \(D_P\), performs Checking Integrity

\(^{17}\) Clients, in turn, need to know the corresponding public keys for verification of quoted bit strings and for encryption that the unbinding key decrypts.
of Auxiliary Information (page 311), and makes a trust assessment based on that information.

Whether clients must trust a cloud’s owner or operators depends on the hosting environment. Trust would be needed if compromising the operation of the server or its state is possible when given physical access to hardware while the server is executing or by privileges granted to administrator accounts or operator consoles. Such information about the hosting environment should be available from description $D_P$ and the associated auxiliary information for the server. Capabilities of administrator accounts and operator consoles would be deduced by analyzing the system software components with descriptors listed in $D_P$. Whether the hardware is tamper-proof would be ascertained from device serial numbers, which ought to be verifiable using the descriptor that $D_P$ lists for each device.\textsuperscript{18}

**Digital Rights Management (DRM).** Computer networks are an attractive infrastructure for distributing digital objects to customers. But certain digital objects require that an access control policy be enforced—no matter where that object is hosted and no matter which hosts in the network are trusted.

- A business that monetizes intellectual property in digital form would want only paying customers to have access to that content, according to various payment plans.

- A business or other institution seeking mandatory access control for digital documents would need to authorize different access for various operations (e.g., view, update, copy, print, or transfer), with privileges assigned according to an employee’s role, identity, prior activity, or other attributes.

An application we trust running on a host we trust can enforce access restrictions on a digital object by incorporating checks in the code it provides to perform operations. A thief could still record sound and images, though, so copyrighted text, music, and video can be stolen—albeit with degradations in fidelity from converting between digital and analog formats. However, active content (i.e., games and software) can be monetized by enforcing restrictions on operations, since the value is in having the capability to interact rather than having a record of the output.

\textsuperscript{18}Descriptions for measured principals will not contain all of the information needed by a client seeking assurance about a hosting environment. For example, to mollify untrusting clients, a cloud provider might enclose each rack of computers in a metal cage having a door that remains locked while those computers are running and for an additional period after a shutdown. Such protection ensures that remnants of confidential data in main memory will decay before it can be read by anyone having physical access to the hardware. Video surveillance of the metal cages completes the defense by deterring employees and others from entering a cage while the processors or memories it encloses still store confidential data that can be read by physical access. Whereas descriptions for measured principals might report serial numbers for hardware, no descriptor would report the locked cages or video cameras in a deployment environment. So actual observation is necessary to have assurance about the physical surroundings for a given device.
Support for measured principals enables hosts and applications that should be trusted to be distinguished from those that should not. That capability, in turn, makes it feasible for decryption keys to be made available only to those trusted hosts and applications. So by disseminating digital objects in encrypted form and requiring all operations on those digital objects to be performed (only) by measured principals, then gating functions for decryption can be used to prevent access on untrusted hosts or by untrusted applications (including hosts and applications that were trusted but subsequently have been modified or had their identities or secrets stolen).

An application that will be trusted to enforce access restrictions for a digital object \(O\) from server \(S\) is implemented as a measured principal (say) \(P\) with a description \(D_P\) and provisioned with an unbinding/binding key pair \(k_P/K_P\). Trust in \(P\) by \(S\) is contingent on \(D_P\) providing assurance that decryption private key \(k_P\) is only available for use by a measured principal consistent with \(D_P\)—modifications to the hosting environment or application code must make \(k_P\) inaccessible. \(S\) and \(P\) can then follow the expected protocol. \(S\) uses remote attestation to check that \(P\) has description \(D_P\) and, therefore, can be trusted. If \(P\) is among those principals \(S\) trusts, then \(S\) encrypts a shared key \(K_O\) using binding public key \(K_P\). Finally, \(S\) sends to \(P\) the encrypted \(K_O\) and the digital object encrypted using \(K_O\). \(P\) uses unbinding key \(k_P\) to recover \(K_S\), which \(P\) uses to decrypt the digital object.

### 11.6.2 Some of the Abuses

Measured principals and gating functions enable software producers—rather than a computer’s owners or operators—to control what programs can be run, what information can be processed, and what programs must be used to process a given digital object. Although we saw above that there are advantages to having such control, it can be abused.

One abuse is software vendor lock-in\(^1\), which occurs when a vendor’s systems are designed to prevent software provided by others from executing on the platform. This practice not only limits competition, but it restricts user-innovation and discourages new entrants to a market. Some argue, though, that software vendor lock-in is a good business model to allow, because it drives technological progress by incentivizing existing vendors to invest in developing new components (since competition is blocked, just like with patent protection).

Another form of abuse arises when DRM is used to automate access control policies today grounded in human judgement. Consider “fair use” which in the United States provides a legal basis for copying excerpts of material protected by copyright. The legal test for “fair use” is nuanced and subjective, making it impossible to implement in a program (even if various machine learning algorithms can mimic human judgement for some settings). So a typical programmed test for “fair use” just bounds the number and/or lengths of excerpts.

---

\(1\) Economists use the term vendor lock-in for situations where a customer is made dependent on a vendor or where the customer would incur substantial costs for switching to another vendor.
That test almost certainly will disallow some copying that a human judge would allow. Another policy that cannot be programmed, so it too is approximated in a manner that is overly-restrictive, is the test for obscenity. A third example is the definition of “fake news” (versus accurate accounts), which seems to require human judgement based on context. In all cases, approximations of a policy must be programmed for use by DRM, these approximations invariably are conservative, and some form of censorship is the result.

Yet there could be benefits to giving software producers control over what software can be executed on a given computer, because such control offers the potential for experts to enhance the security of individual systems and networks. We see this when software for a platform must be downloaded from a platform vendor’s “app store” that offers software the vendor has evaluated and found not to have certain vulnerabilities. Most owners or operators of a computer would be incapable of undertaking such an evaluation and, therefore, would happily delegate that task and responsibility to some software producers. So an owner or operator avoids the risks of running insecure software by having a computer that blocks software not downloaded from the “app store”. Moreover, networks that include this computer benefit too, because insecure machines in a network can be co-opted by attackers and used as platforms for attacking other machines in the network.

Critics of measured principals and gating functions complain about a transfer to system developers of rights that previously were held by computer owners and operators. But measured principals and gating functions also transfer responsibilities. When a computer is connected to a network, its owners and operators have a responsibility to ensure the computer does not attack computers elsewhere in the network. With measured principals and gating functions, this responsibility is delegated to the system developer. A more-capable party now controls what software can be downloaded and run. So measured principals and gating functions make it unnecessary to invest in educating the ever growing number of computer owners and operators.

Notes and Reading

The abstractions introduced in this chapter evolved from efforts in the late 1980’s at Digital Equipment Corporation to build secure distributed systems assuming secure stand-alone systems. Gasser et al. [16] gives an overview, describing an architecture where a user’s login at one computer suffices for accessing objects hosted at other computers. In this architecture, the cryptographic hash of a software system’s binary is the name for the principal identified with

\[20\] As example of such a policy is Supreme Court Justice Potter Stewart’s widely quoted 1964 test from Jacobellis v. Ohio: “I know it when I see it”. Originally offered as a test for hard-core pornography, it is now often cited as a test for obscenity.

\[21\] Beware: An app store that claims to perform evaluations might not do so, might not be thorough, might impose arbitrary restrictions, and/or it might exist primarily to block offerings from competitors.
execution of that binary, since this name can be checked to determine if the expected is running; authenticated messages are proposed for attributing requests made by software running on a remote host, so such requests can be authorized; and remote attestation was introduced for deciding whether a remote computer should be trusted to perform operations or to hold sensitive data. The protocol sketched §11.5.2 for remote attestation at system startup was first described there.

Fast forward a decade. The PC, which is running Microsoft’s Windows operating system on an x86 microprocessor, has become ubiquitous. It is not secure. New applications—for example, disseminating copyrighted digital content and offering on-line financial services—will depend critically on enforcing confidentiality and integrity. Other market growth, too, has slowed because computers cannot be trusted with data or operations. So Microsoft embarked on a project to build a PC-based platform that would be secure. Next-Generation Secure Computing Base\(^\text{22}\) (NGSCB) \[^{9}\] would provide strong isolation for new applications, protect cryptographic keys and other secrets, yet still run legacy Windows applications. To achieve these goals, the new platform would involve new hardware support and a new operating system.

The NGSCB design enforced isolation by using several mechanisms. A virtual machine manager supported “red” virtual machines that ran legacy Windows and “green” virtual machines that ran a new operating system for applications requiring higher assurance. Hardware support for \textit{curtained memory} \[^{10}\] would ensure that access to certain memory locations could be made only by code in a specified address range. And there would be hardware support for gating functions, a term introduced by England and Peinado in a paper \[^{12}\] written to show that the construct had utility beyond NGSCB.

Although NGSCB never became a Microsoft product, some of its elements have been incorporated into Windows releases and doubtless others will be. BitLocker full disk encryption (which inspired the discussion in §11.6) appears in Windows Vista; secure boot and trusted boot appears in Windows 8; and device guard, which uses a separate virtual machine for isolation, appears in Windows 10.

The anticipation of NGSCB also stimulated a policy debate (distilled in §11.6) about rights accompanying ownership of platforms and digital content, potential abuses \[^{1, 2, 31}\] when those rights are delegated and/or restricted, potential benefits \[^{26}\] of requiring delegation, and potential drawbacks \[^{17}\] when users cannot share new products and services they develop for themselves.

A secure co-processor, called the \textit{trusted platform module} (TPM), would provide NGSCB with the needed hardware support for gating functions to support sealing, quoting, and unbinding. The TPM specification was developed by the Trusted Computing Platform Alliance\(^\text{23}\) (TCPA), formed in 1999 for this purpose by Compaq\(^\text{24}\), HP, IBM, Intel, and Microsoft. (Other companies have since

\[^{22}\]The initial name was Palladium. Microsoft changed the name in January 2003.

\[^{23}\]Trusted Computing Platform Alliance was renamed Trusted Computing Group (TCG) in April 2003.

\[^{24}\]In June 1998, Digital Equipment Corporation, on the brink of insolvency, had been ac-
The material in §11.3 describes a simplified TPM.\textsuperscript{25} See the Trusted Computing Group’s website \cite{33} for the full TPM specification.

Anticipating that TPMS or other hardware to support gating functions could become widely available, researchers started investigating how best to leverage such functionality. Terra \cite{15} was among the first efforts, implementing a hypervisor that supports two kinds of virtual machines. An \textit{open box} virtual machine provides a standard instruction set architecture and was intended for enforcing isolation of a legacy operating system running a few, cooperating, applications. A \textit{closed box} virtual machine provides capabilities to perform remote attestation and sealing, thereby protecting applications from other execution on that computer and from the platform’s owners, operators, or anyone else with access to disks or other state stored outside of the processor.

The instruction set of virtual machines implemented by Terra (and NGSCB) does not include a virtual TPM. There are good reasons for this design choice. Although a virtual machine manager could create a chain of trust from a virtual TPM to an underlying hardware TPM, the semantics of the resulting virtual TPM would not be the same as a hardware TPM—a root of trust implemented by software can be compromised in ways that a hardware implementation cannot. Virtual TPMS, nevertheless, can be useful. So Berger et al. \cite{5} extends Xen \cite{3} to provide each virtual machine with a TPM. Other approaches to realizing virtual TPMS have also been explored. Para-virtualization to realize a version 1.2 TPM is discussed in England and Loeser \cite{11}; Yap and Tomlinson \cite{37} discuss para-virtualization for realizing a version 2.0 TPM.

CloudProxy \cite{20,32}, which is intended for deployment in a cloud datacenter (and is the basis for the cloud-hosted services implementation sketched in §11.6), also offers a para-virtualization. The so-called “CloudProxy Tao” prescribes that each level of a software stack provide gating functions for sealing, quoting, and unbinding. A cloud-based application would be run in an isolated virtual machine. Within that virtual machine, the operating system and the application each would use the gating functions available to that layer in order to protect information stored on disks or conveyed in messages.

Sailer et al. \cite{25}, explores other operating system support for attestation of a software stack. In this work, a Linux kernel was modified to make and append \textit{integrity measurements} onto \textit{measurement lists} that are updated if an executable is loaded or if a (new) \textit{measure} system call is invoked, naming a file to be measured. The integrity of measurement lists is protected by cryptography using keys stored in a TPM. Using the vocabulary of this chapter, Sailer et al. \cite{25} extends Linux to implement measured principals that are characterized by descriptions comprising lists of descriptors. Remote attestation of these measured principals is supported, but gating functions for sealing, quoting, and unbinding are not.

In this chapter, as in much of the literature, authorization for gating functions is based on names of principals, where names are hashes of executables required by Compaq, a PC manufacturer.\textsuperscript{25} That discussion benefitted greatly from Ariel Segall’s creative commons video course \cite{27} about TPMS.
and/or files. Since an executable must be analyzed to infer properties of its executions, the connection between a principal’s name and its properties is indirect. The goals of gating functions are better served if that connection can be direct. With property-based attestation [23, 24], the name for a measured principal satisfying a property is formed by computing a hash for the property description. The descriptors for properties discussed on page 310 exemplify this. Sadeghi and Stiible [23] discusses having property descriptions be certificates from trusted third parties, and their subsequent work with Winandy [24] avoids third-party dependence by delegating to analyzers that are part of the system.

The Nexus [26] [29] operating system disentangles property attestation from the naming of principals. Specifically, Nexus provides a unified language for authorization decisions that combine attestations of axiomatic, analytic, and synthetic properties of a principal and/or its execution environment. Nexus also offers operating system support for a form of capabilities based on statements from a TPM. To gain experience with the approach, a social networking system that enforces fair resource allocation, code safety, and confidentiality was implemented on Nexus. Other applications Nexus supported include a video player that enforces DRM and the NetQuery [28] system for managing a knowledge plane in a network.

The thread of work that started with NGSCB goes today by the label “Trusted Computing”. It includes research and development both in software and in hardware. Mitchell [21] collects papers about Trusted Computing research circa 2005, and Parno et al. [22] offers an excellent survey about system support for collecting and conveying information for making trust assessments. Various forms of hardware support for Trusted Computing are now also becoming available. Typically, the goal is support for functionality that is believed easier to use than gating functions or the TPM’s packaging of them. The secure enclaves supported by Intel’s Software Guard Extensions (SGX) [18] is a notable example. So far, though, SGX has not been widely embraced—rich abstractions can involve a steep learning curve and tend to exhibit implementation vulnerabilities. Costan and Devadas [6] documents vulnerabilities in initial releases of SGX. Secure enclaves are also supported by Sanctum [7] and Komodo [14]—but with part of that functionality implemented as software.

Cryptography is practical for enforcing isolation only if (i) delays to perform encryption/decryption are not problematic and (ii) access to cryptographic keys is controlled. Hardware-implemented gating functions provide one solution, but other hardware support has also been suggested. In 1979, for example, Dorothy Denning [8] proposed (and later defended [19]) that an RSA public-key encryption/decryption device be inserted between each workstation and a remote file server in a distributed system. Encryption would protect information that is in transit or stored on the file server; an individual’s cryptographic keys would be kept (only) in a removable ROM chip, carried by that individual and plugged into the encryption/decryption device while operating the workstation.

26Nexus also is the name of a component in Microsoft’s NGSCB. These are different.
With their Citidal architecture [36], IBM researchers circa 1991 turned their attention to security in distributed systems comprising devices located where attackers had physical access. Earlier IBM research efforts μABYSS [34] and ABYSS [35] had explored hardware that obstructed physical access to hardware for cryptographic operations and storing keys. Building on this work, Citidal’s architecture would protect programs and information by combining safe environments (like restricted-access machine rooms), robust enclosures, and cryptography.

Citidal’s thesis was simple, yet compelling: Information may exist as plaintext on a computer located within safe environment or on hardware having a robust physical enclosure; otherwise, the information must be encrypted. Coupled with a belief\(^\text{28}\) that performing operations on encrypted data was not possible, Citidal’s thesis determined where specialized cryptographic hardware must be deployed and when information must be encrypted. To put this vision into commercial practice required suitable cryptographic co-processors. IBM did take this next step, translating what was learned with Citidal into the IBM 4758 series cryptographic co-processors, which reached the market in 1997 [30].

Cryptographic hardware is useful even when physical security is not a focus. Assurance for a platform requires some basis to believe that the intended boot and IPL routines are what was executed (and not other code). Gasser et al. [16] described the basic scheme for obtaining such assurance when a system comprises a stack of software layers: Each layer authenticates the code for its successor before transferring control there, where authentication of a layer involves checking a hash and/or performing decryption. A secure hardware cryptographic co-processor is thus ideal for inclusion in the lowest layer. Arbaugh et al. [13] addresses the rich case of an “open” architecture (as found in the PC). Here, transfers of control during system startup are better characterized as a star rather than as a stack, because call/return transfers of control are used to execute initialization for various peripheral devices.

Bibliography


[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art

\(^{27}\)ABYSS is an acronym for A Basic Yorktown Security System. The research group was located at IBM Research in Yorktown Heights, NY.

\(^{28}\)This belief is no longer completely valid. Progress on homomorphic encryption means that certain computations with encrypted data now can be performed.


