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., `Linux4.8.0-36-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 implement this approach.

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 the name of a measured principal 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 properties we believe to be satisfied by the principal's executions.

A *gating function*[1] $[K\text{-}\mathbb{F}](\cdot)$ is a cryptographic function obtained by adding access control that restricts uses of cryptographic function $K\text{-}\mathbb{F}(\cdot)$ and its associated key K:

- Key K can be used only by gating functions $[K\text{-}\mathbb{F}](\cdot)$ naming K. Other accesses to K are blocked.

[1] The use of square brackets for naming a gating function is not standard notation. We introduce it to distinguish gating functions from ordinary cryptographic functions, graphically depicting that access to K and $\mathbb{F}$ is protected.

- Invocations of $K\text{-}\mathbb{F}(\cdot)$ succeed only in systems satisfying an associated *configuration constraint* $Config([K\text{-}\mathbb{F}])$, 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. That means $Config([K\text{-}\mathbb{F}])$ can provide assurance about whether invocations of $[K\text{-}\mathbb{F}](\cdot)$ could violate a given security policy because (i) an invocation of $[K\text{-}\mathbb{F}](\cdot)$ requires that the measured principals in $Config([K\text{-}\mathbb{F}])$ have started executing, and (ii) the name of a measured principal can be associated with 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 enforced by the system abstractions—processes, virtual machines, and containers—discussed in Chapter 10. The isolation enforced by gating functions is weaker because they don't protect availability, whereas the system abstractions block unauthorized writes and, therefore, do preserve availability. The isolation enforced by gating functions is stronger because they 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 run 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[2] $\blacktriangleleft d_1\, d_2\, \ldots\, d_n \blacktriangleright$ of *descriptors* such that[3]

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

[2] Using 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.

[3] For 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.

- 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 goal is for a description to indicate whether the associated measured principal can be trusted to satisfy some properties. This goal is facilitated if each descriptor is not just an arbitrary name but 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 $\blacktriangleleft d_1\, d_2\, \ldots\, d_n \blacktriangleright$ 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 facilitate blocking 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 different name would cause gating functions to block P' from using the cryptographic keys that P is using. 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 Sealed-State Upgrade Protocol (page 309) 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 $[K\text{-}\mathbb{F}](\cdot)$ 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 $\mathit{Config}([K\text{-}\mathbb{F}])$. 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 will not co-opt a gating function if it is infeasible to predict the name $\mathcal{N}(D)$ that will be assigned to that measured principal. The following requirements on naming function $\mathcal{N}(\cdot)$ provide that defense.

> **Names for Measured Principals.** Function $\mathcal{N}(\cdot)$ for assigning name $\mathcal{N}(D)$ to a measured principal having description D satisfies:
>
> *Collision Resistance.* If $D \neq D'$ holds then, with high probability, $\mathcal{N}(D) \neq \mathcal{N}(D')$ holds.[4]
>
> *Preimage Resistance.* Given D, it is infeasible to construct a description D' where $D \neq D'$ and $\mathcal{N}(D) = \mathcal{N}(D')$ hold. □

[4] Names will be fixed length and relatively short, whereas descriptions are neither. So there are more possible descriptions than names, and mapping $\mathcal{N}(\cdot)$ 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 $\mathcal{N}(D)$ by creating a principal with description D' that satisfies $\mathcal{N}(D') = \mathcal{N}(D)$.

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

$$\mathcal{N}(D) = \begin{cases} 0 & \text{if } D = \blacktriangleleft\ \blacktriangleright \\ \mathbb{H}(\mathcal{N}(\ \blacktriangleleft d_1\ \ldots\ d_{n-1}\blacktriangleright\ ) \cdot \mathbb{H}(\ d_n\ )) & \text{if } D = \blacktriangleleft d_1\ d_2\ \ldots\ d_n\blacktriangleright \end{cases} \tag{11.1}$$

By unwinding the recursion in (11.1), we see that $\mathcal{N}(\blacktriangleleft d_1\ d_2\ \ldots\ d_n\blacktriangleright)$ incorporates the descriptors of $\blacktriangleleft d_1\ d_2\ \ldots\ d_n\blacktriangleright$ in order of appearance, and there is a function to generate the name for a description $\blacktriangleleft d_1\ d_2\ \ldots\ d_n\ d\blacktriangleright$ by using the name N for a description $\blacktriangleleft d_1\ d_2\ \ldots\ d_n\blacktriangleright$:

$$\mathcal{X}(N, d)\text{:}\quad \mathbb{H}(\ N \cdot \mathbb{H}(\ d\ )\ ) \tag{11.2}$$

We thus say that $\mathcal{X}(N, d)$ *extends* name N with descriptor d.

Allowing $\mathcal{N}(D)$ to be calculated incrementally accommodates descriptions incorporating 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 $\mathcal{N}(D)$ 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 access to the resource corresponding to d_i is needed for deciding subsequent execution.

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. 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 $\mathcal{N}(D_P) = N_P$ 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 $\mathbb{H}(\ B_O\ )$.

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 is undesirable then O might be represented as a pair $\langle ARM_O, AID_O \rangle$, where ARM_O is an address-relocation map and AID_O is an address-independent description. Hash $\mathbb{H}(AID_O)$ can then serve as 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 $\mathcal{N}(hw)$ then using $\mathbb{H}(\mathcal{N}(hw))$ 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 unique 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 $\mathcal{N}(hw)$ for a hardware processor hw from a unique signing key k^{id}_{hw} that hw stores, uses, but never reveals. Tamper-proof packaging prevents attackers who have physical access to hw from learning or changing k^{id}_{hw}. Key k^{id}_{hw} could be generated and installed by the manufacturer of hw when the device is fabricated[5] or the device itself could generate k^{id}_{hw} by, for example, leveraging fixed and unique timing differences inevitably present in a specific semiconductor chip.

A processor instruction to compute k^{id}_{hw}-signed messages then allows corresponding verification key K^{id}_{hw} to serve as $\mathcal{N}(hw)$. To make this hardware processor naming scheme concrete, code for checking whether $\mathcal{N}(hw) = K$ holds is given in Figure 11.1.[6] The routine assumes that a register $\texttt{qkr}_{\text{id}}$ comes preloaded with k^{id}_{hw}. It also assumes this register cannot be accessed except by executing instruction[7] $\texttt{quote}(\texttt{qkr}_{\text{id}}, r, out)$, which computes k^{id}_{hw}-$\mathbb{S}(r)$ and writes that value into memory out. Therefore, if r is fresh and $AuthHW(K, r)$ returns *true* then $\mathcal{N}(hw) = K$ holds and the underlying processor has name K.

When needed, information about the instruction set architecture ISA_{hw} that the processor with name K^{id}_{hw} implements would be conveyed using a certificate

$$k_C\text{-}\mathbb{S}(K^{\text{id}}_{hw} \textbf{ speaksfor } ISA_{hw}) \tag{11.3}$$

[5] A manufacturer should not keep records of k^{id}_{hw}. Otherwise, attackers could learn k^{id}_{hw} by compromising the database being stored at the manufacturer.

[6] K-$\mathbb{S}?(m)$ used in Figure 11.1 evaluates to *true* if m is digitally signed by a private key corresponding to verification key K.

[7] See §11.3.3 for a detailed discussion of the `quote` instruction.


```
AuthHW: function(K, r)
    quote(qkr_id, r, out)
    if K-𝕊?(out)
      then return( true )
      else return( false )
    end AuthHW
```

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

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_{hw}^{id} **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 the 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. After such an upgrade, 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. Some means also would be employed to revoke the old certificate.

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.

One solution is for a processor hw to have one or more *attestation identity keys*, each a surrogate for K_{hw}^{id} and usable as the name for hw. Validity of an attestation identity key K_{hw}^{A} is conveyed by a certificate

$$k_T\text{-}\mathbb{S}(K_{hw}^{\text{A}} \textbf{ speaksfor } ISA_{hw}) \tag{11.4}$$

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

$$k_C\text{-}\mathbb{S}(K^{\mathrm{id}}_{hw}\ \textbf{speaksfor}\ ISA_{hw}),\ \ k^{\mathrm{id}}_{hw}\text{-}\mathbb{S}(K^{\mathrm{A}}_{hw}\ \textbf{speaksfor}\ K^{\mathrm{id}}_{hw})$$

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 identify 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]

[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*.

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 crucial. But additional mechanism is not needed for protecting that integrity, provided the integrity of measured principal names is being protected.

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).
   (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* $\texttt{skr}_1$, ..., $\texttt{skr}_N$, *quoting-key registers* $\texttt{qkr}_{id}$, $\texttt{qkr}_1$, ..., $\texttt{qkr}_N$, and *unbinding-key registers* $\texttt{ukr}_1$, ..., $\texttt{ukr}_N$.
- Instructions that use a key K in a specified key register and compute a gating function $[K\text{-}\mathbb{F}](\cdot)$ to perform
  - *sealing* to protect the confidentiality and integrity of local content,
  - *quoting* to establish authenticity of locally produced content, or
  - *binding* to import remote content but only if the local system satisfies certain configuration constraints,

  where $Config([K\text{-}\mathbb{F}])$ 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).

11.3.1 Measurement Registers and Constraints

Measurement registers $\texttt{mr}_0$, $\texttt{mr}_1$, ..., $\texttt{mr}_N$ are used for defining configuration constraints.

Measurement register $\texttt{mr}_0$ 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.

[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.

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 $\texttt{mr}_1$, $\texttt{mr}_2$, ..., $\texttt{mr}_N$ (unlike $\texttt{mr}_0$) 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.

Instruction	Operation
$\texttt{MRreset}(\texttt{mr}_i)$	$\texttt{mr}_i := 0$
$\texttt{MRextend}(\texttt{mr}_i, mem)$	$\texttt{mr}_i := \mathbb{H}(\texttt{mr}_i \cdot \mathbb{H}(mem))$

`MRreset` and `MRextend` are the only instructions that change $\texttt{mr}_1$, $\texttt{mr}_2$, ..., $\texttt{mr}_N$. 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 $\texttt{mr}_1$, $\texttt{mr}_2$, ..., $\texttt{mr}_N$ ideally suited to storing names of measured principals.

Configuration Constraints. A configuration constraint $\mathcal{C}$ will be specified by listing a subset of the measurement registers and, for each, giving a value it must store. Set of pairs

$$\mathcal{C} \;=\; \{\ldots, \langle i, v_i\rangle, \ldots\}$$

specifies configuration constraint

$$\bigwedge_{\langle i,v_i\rangle \in \mathcal{C}} \texttt{mr}_i = v_i. \tag{11.5}$$

and can involve any subset of the measurement registers.

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

$$\mathit{ConfigSat}(\mathcal{C}):\ \langle i, v_i\rangle \in \mathcal{C} \;\Rightarrow\; \texttt{mr}_i = v_i$$

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 $\texttt{mr}_1$ contain the name of a measured principal

for the operating system, $\texttt{mr}_2$ 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 `MRextend` 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 build a *chain of trust* by following execution back through the operating system, boot-loader, firmware, and ultimately to 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 $K/\mathcal{C}$*-sealed* bit strings, where K is a symmetric key stored in some *sealing-key register*, and $\mathcal{C}$ is a configuration constraint.

- Reading a $K/\mathcal{C}$-sealed bit string *sb* reveals nothing about K or about the unsealed bit string from which *sb* was derived.

- Updates to a $K/\mathcal{C}$-sealed bit string are not blocked but cause subsequent execution of `unseal` to fail.

Because writes to $K/\mathcal{C}$-sealed bit strings are not blocked, availability of a $K/\mathcal{C}$-sealed bit string is compromised by writing to it but integrity of a $K/\mathcal{C}$-sealed bit string 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 those measurement registers selected by a bit string *crSet* string stored in memory.[10] Each sealing key register $\texttt{skr}_i$ comprises two fields: $\texttt{skr}_i\texttt{.key}$, which stores the key, and $\texttt{skr}_i\texttt{.config}$, which stores the configuration constraint.

Instruction	Operation
$\texttt{SKRgen}(\texttt{skr}_i, crSet)$	$\texttt{skr}_i\texttt{.key} :=$ fresh symmetric key $\texttt{skr}_i\texttt{.config} := \{\langle i, v_i\rangle \mid crSet[i] \wedge \texttt{mr}_i = v_i\}$

Instruction `seal` is a gating function for shared-key authenticated encryption function $K\text{-}\mathbb{E}^{\mathbb{A}}(\cdot)$; instruction `unseal` is a gating function for corresponding decryption function $K\text{-}\mathbb{D}^{\mathbb{A}}(\cdot)$, which fails rather than decrypting an argument

[10] As usual, 1 denotes *true* and 0 denotes *false*. So a measurement register $\texttt{mr}_i$ is selected if and only if $crSet[i] = 1$ holds.

not produced by K-$\mathbb{E}^{\mathbb{A}}(\cdot)$.[11]

Instruction	Operation
$\texttt{seal}(\texttt{skr}_i, in, out)$	out := $\texttt{skr}_i.\texttt{key-}\mathbb{E}^{\mathbb{A}}(in)$
$\texttt{unseal}(\texttt{skr}_i, in, out)$	**if** *ConfigSat*$(\texttt{skr}_i.\texttt{config})$ **then** out := $\texttt{skr}_i.\texttt{key-}\mathbb{D}^{\mathbb{A}}(in)$ **else fail**

Any program can invoke `seal` to encrypt information—independent of the system 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-$\mathbb{D}^{\mathbb{A}}(in)$ then fails).

By sealing the state that is being saved between executions of a given program or service, we protect the confidentiality of that state from attackers—including across system reboots. 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 can be 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

[11] One implementation for K-$\mathbb{E}^{\mathbb{A}}(\cdot)$ and K-$\mathbb{D}^{\mathbb{A}}(\cdot)$, 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-$\mathbb{E}^{\mathbb{A}}(\cdot)$ returns:

$$K\text{-}\mathbb{E}^{\mathbb{A}}(m) = \langle K_1\text{-}\mathbb{E}(m),\ \mathbb{H}(K_1\text{-}\mathbb{E}(m) \cdot K_2)\rangle.$$

K-$\mathbb{D}^{\mathbb{A}}(\langle x, h\rangle)$ returns K_1-$\mathbb{D}(x)$ only if $h = \mathbb{H}(x \cdot K_2)$ holds; otherwise the invocation of K-$\mathbb{D}^{\mathbb{A}}(\langle x, h\rangle)$ fails. See Bellare and Namprempre [4] for strengths and weakness of alternative constructions.

key. This two-level scheme, however, has risks. A cryptographic key now would be 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, $\texttt{kr}_\texttt{j}$ denotes an arbitrary key register, *krSet* is a bit string that selects a subset of the key registers, and *ska* is the memory address for the key archive.

Instruction	Operation
$\texttt{KRseal}(\texttt{skr}_i, krSet, ska)$	**let** $keyArchive = \{\langle j, v_j\rangle \mid krSet[j] \wedge \texttt{kr}_j = v_j\}$ **in** $ska := \texttt{seal}(\texttt{skr}_i, keyArchive)$
$\texttt{KRunseal}(\texttt{skr}_i, ska)$	**let** $keyArchive = \texttt{unseal}(\texttt{skr}_i, ska)$ **in for** $\langle j\ val_j\rangle \in keyArchive$ **do** $\texttt{kr}_j := val_j$

Because a key archive is a sealed bit string, reading a key archive reveals nothing about keys or configuration constraints it contains. In addition, writing 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 $\texttt{qkr}_{\text{id}}$ has a fixed value and has no associated configuration constraints

    $$\texttt{qkr}_{\text{id}}.\texttt{key} = k^{\text{id}}_{hw} \qquad\qquad \texttt{qkr}_{\text{id}}.\texttt{config} = \varnothing$$

    where k^{id}_{hw} is a unique signing key associated with the processor *hw* that hosts this register.[12]

[12] Recall (see page 302), corresponding verification key K^{id}_{hw} will be well known.

  - Quoting-key registers $\texttt{qkr}_1$, ..., $\texttt{qkr}_N$ can be reloaded.

- Verification keys for checking digital signatures and the certificates to identify those verification keys are not secret and, therefore, they are stored in memory rather than being stored in a key register.

`QKRgen` provisions a quoting-key register by (i) generating a fresh signing key, (ii) associating a specified configuration constraint, and (iii) storing in memory a k^{id}_{hw}-signed certificate that gives the verification key for signatures generated using this fresh signing key.

Instruction	Operation
`QKRgen`($\texttt{qkr}_i$, *crSet*, *mem*)	$\texttt{qkr}_i.\texttt{config} := \{\langle i, v_i\rangle \mid crSet[i] \wedge \texttt{mr}_i = v_i\}$ **let** k/K be a fresh private/public key pair **in** $\texttt{qkr}_i.\texttt{key} := k$ $mem := \texttt{qkr}_{\text{id}}.\texttt{key-}\mathbb{S}(\texttt{qkr key: } i \mid K)$

Instruction `quote` is a gating function based on a cryptographic function k-$\mathbb{S}(\cdot)$ for generating digitally-signed bit strings.

Instruction	Operation
`quote`($\texttt{qkr}_i$, *in*, *out*)	**if** *ConfigSat*($\texttt{qkr}_i.\texttt{config}$) **then** $out := \texttt{qkr}_i.\texttt{key-}\mathbb{S}(\texttt{sig: } i \mid in)$ **else fail**

Notice, bit strings that `quote` produces cannot be mistaken for certificates that `QKRgen` produces—each contains a different prefix. As we shall see, each kind of instruction 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.

Retrieving Configurations. To trust results produced by executing a gating function $[K\text{-}\mathbb{F}](\cdot)$ requires knowledge of the configuration constraint associated with the key register that stores K. Instruction `KRgetConf` provides that information; it stores into memory a k^{id}_{hw}-signed certificate for the configuration constraint currently associated with a specified key register. A second instruction `KRgetCurConf` reveals the current configuration, so a program can test whether a key register's configuration constraint is currently satisfied; `KRgetCurConf` stores into memory a k^{id}_{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.

Instruction	Operation
`KRgetConf`($\texttt{kr}_i$, r, *out*)	$out := \texttt{qkr}_{\text{id}}.\texttt{key-}\mathbb{S}(\texttt{keyCnfig: } i \mid r \mid \texttt{kr}_i.\texttt{config})$
`KRgetCurConf`(*crSet*, r, *out*)	$cc := \{\langle i, v_i\rangle \mid i \in crSet \wedge \texttt{mr}_i = v_i\}$ $out := \texttt{qkr}_{\text{id}}.\texttt{key-}\mathbb{S}(\texttt{curCnfig: } r \mid cc)$

Certificates from KRgetCurConf that include $\texttt{mr}_0$ in *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 a descriptor for the object. A measured principal using an outdated version of the data object then would 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 provided by a gating function $[k\text{-}\mathbb{D}](\cdot)$ then information that is encrypted under the corresponding public key K can be recovered only by systems satisfying configuration constraint $\mathit{Config}([k\text{-}\mathbb{D}])$. By encrypting some information under public key K, we are *binding* that information to the specific system that is able to use gating function $[k\text{-}\mathbb{D}](\cdot)$ to invert that encryption. Gating function $[k\text{-}\mathbb{D}](\cdot)$ 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 UKRgen to generate a public/private key pairs with appropriate certificates, and an instruction UKRdec that implements a gating function for decryption using a private key stored in some unbinding-key register.

Instruction	Operation
UKRgen($\texttt{ukr}_i$, *crSet*, *mem*)	$\texttt{ukr}_i.\texttt{config} := \{\langle i, v_i\rangle \mid \mathit{crSet}[i] \wedge \texttt{mr}_i = v_i\}$ **let** k/K be a fresh private/public key pair **in** $\texttt{ukr}_i.\texttt{key} := k$ $\mathit{mem} := \texttt{qkr}_{\mathrm{id}}.\texttt{key-}\mathbb{S}(\texttt{ukr key: } i \,\texttt{I}\, K)$
UKRdec($\texttt{ukr}_i$, *in*, *out*)	**if** $\mathit{ConfigSat}(\texttt{ukr}_i.\texttt{config})$ **then** $\mathit{out} := \texttt{ukr}_i.\texttt{key-}\mathbb{D}(\mathit{in})$ **else fail**

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

1. $R \longrightarrow S$: $\langle r, P \rangle$, where r is a fresh nonce.
2. S: Generate fresh public/private keys $K_P^{\text{att}}/k_P^{\text{att}}$ for use with a gating function $[k_P^{\text{att}}\text{-}\mathbb{S}](\cdot)$, where $\mathit{Config}([k_P^{\text{att}}\text{-}\mathbb{S}]) = \{P\}$.
3. $S \longrightarrow R$: $[k_S^{\text{att}}\text{-}\mathbb{S}](r, P, K_P^{\text{att}})$
4. R: Accept K_P^{att} 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_S^{att} verifies digital signature received step 3.

Figure 11.2: Remote Attestation by R of P Running on S

11.4 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_P^{att} for verifying signatures on messages digitally signed by P.

To decide whether P and messages that K_P^{att} verifies can be trusted, it suffices to have a candidate description D_P along with associated auxiliary information, perform Checking Integrity of Auxiliary Information (page 305), and then engage in the necessary analysis to determine that the system D_P describes satisfies expected properties, including that digital signatures verified by K_P^{att} can be created only by executing P.

Note, if rebooting the processor that is running P does not delete signing key k_P^{att} then clients need to defend against TOCTOU attacks involving a reboot that is instigated between the remote attestation that K_P^{att} provided and the subsequent sending of messages that K_P^{att} verifies. One such defense is to include in D_P a descriptor for a value that is incremented with each reboot. The value of $\texttt{mr}_0$ would work; but a certificate giving the current time and signed by a trusted source also would work.

11.4.1 A Remote Attestation Protocol

A remote attestation protocol is sketched in Figure 11.2. By communicating with some measured principal S, initiator R learns a name P and attestation public key K_P^{att} for some measured principal that S is executing. The protocol depends on the following assumptions.

(i) R trusts S and has an attestation public key K_S^{att} 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 $[k_P^{\text{att}}\text{-}\mathbb{S}](\cdot)$, for generating digital signatures, as required in step 2 of the protocol in Figure 11.2.

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 `KRgetConf` to obtain a k_{hw}^{id}-signed certificate that attests $\mathit{Config}([k_S^{\text{att}}\text{-}\mathbb{S}])$ is the configuration constraint associated with a gating function $[k_S^{\text{att}}\text{-}\mathbb{S}](\cdot)$ and K_S^{att} is the associated verification key. This certificate is returned to R. Knowledge of verification key K_{hw}^{id} 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_{hw}^{id}-signed certificate specifies. R then concludes K_S^{att} **speaksfor** 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 11.2 is for initiator R to obtain values for P and K_P^{att} satisfying

$$K_P^{\text{att}} \textbf{ speaksfor } P. \tag{11.6}$$

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

Step 3 of the protocol in Figure 11.2, is formalized as

$$K_S^{\text{att}} \textbf{ says } (S.r \textbf{ says } (K_P^{\text{att}} \textbf{ speaksfor } P)) \tag{11.7}$$

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

$$S.r \textbf{ speaksfor } S. \tag{11.8}$$

Assumption (i) above implies

$$K_S^{\text{att}} \textbf{ speaksfor } S. \tag{11.9}$$

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

$$\text{GATING:}\frac{\{T\} = \mathit{Config}([k_T^{\text{att}}\text{-}\mathbb{S}])}{K_T^{\text{att}} \textbf{ speaksfor } T}\ . \tag{11.10}$$

The hypothesis $\{T\} = \mathit{Config}([k_T^{\text{att}}\text{-}\mathbb{S}])$ is discharged by obtaining a k_{hw}^{id}-signed certificate from S, as discussed above.

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

$$S \textbf{ says } (S.r \textbf{ says } (K_P^{\text{att}} \textbf{ speaksfor } P)).$$

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

$$S.r \textbf{ says } (S.r \textbf{ says } (K_P^{\text{att}} \textbf{ speaksfor } P)) \tag{11.11}$$

CAL inference rule SAYS-E with (11.11) infers:

$$S.r \textbf{ says } (K_P^{\text{att}} \textbf{ speaksfor } P)$$

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 SUBPRIN we conclude S **speaksfor** P, and by transitivity of **speaksfor** CAL inference rule DELEG-TRANS with (11.8) we get $S.r$ **speaksfor** P. Applying (9.16) we get:

$$P \textbf{ says } (K_P^{\text{att}} \textbf{ speaksfor } P)$$

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

11.4.2 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 does require that system startup undertake advance preparation for servicing future attestation requests.

System startup typically comprises *stages* that 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 $D_S^0, D_S^1, \ldots, D_S^n$ of descriptions. Each D_S^i is the description for a stage and corresponds to a measured principal $\mathcal{N}(D_S^i)$, with D_S^n satisfying $\mathcal{N}(D_S^n) = S$. A *system startup remote attestation protocol*

- associates a pair of public/private attestation keys $K_i^{\text{att}}/k_i^{\text{att}}$ with each stage $\mathcal{N}(D_S^i)$,
- restricts uses of k_i^{att} according to $\mathit{Config}([k_i^{\text{att}}\text{-}\mathbb{S}]) = \{\mathcal{N}(D_S^i)\}$, and
- provides a set $\mathit{AttCerts}_S$ of certificates from which K_i^{att} **speaksfor** $\mathcal{N}(D_S^i)$ can be inferred.

1. $k_0^{\text{att}} := k_{hw}^{\text{id}}$; $K_0^{\text{att}} := K_{hw}^{\text{id}}$; $\mathcal{N}(D_S^0) := \mathcal{N}(hw)$;
2. **for** $i := 0$ **to** $n-1$ **do**
   (a) $\mathcal{N}(D_S^i)$ loads software for its successor stage, creating description D_S^{i+1}. Compute name $\mathcal{N}(D_S^{i+1})$.
   (b) $\mathcal{N}(D_S^i)$ obtains public/private attestation keys $K_{i+1}^{\text{att}}/k_{i+1}^{\text{att}}$ and installs a gating function $[k_{i+1}^{\text{att}}\text{-}\mathbb{S}](\cdot)$, with $\mathit{Config}([k_{i+1}^{\text{att}}\text{-}\mathbb{S}]) = \{\mathcal{N}(D_S^{i+1})\}$.
   (c) $\mathit{AttCerts}_S := \mathit{AttCerts}_S \cup \{k_i^{\text{att}}\text{-}\mathbb{S}(K_{i+1}^{\text{att}}, \mathcal{N}(D_S^{i+1}))\}$
   (d) $\mathcal{N}(D_S^i)$ relinquishes control to $\mathcal{N}(D_S^{i+1})$ if $\mathit{AuthStages}(\mathcal{N}(D_S^{i+1}))$ holds; otherwise $\mathcal{N}(D_S^i))$ halts.

Figure 11.3: System Startup Attestation Protocol

One such protocol is given in Figure 11.3. The resulting set of certificates $\mathit{AttCerts}_S$ can be used to construct a chain for justifying trust in measured principal $\mathcal{N}(D_S^n)$. In step 2a of the protocol, a stage $\mathcal{N}(D_S^i)$ loads its successor $\mathcal{N}(D_S^{i+1})$. For step 2b, two approaches could ensure that private attestation key k_{i+1}^{att} has not previously been revealed: (i) generate a fresh k_{i+1}^{att}, or (ii) use key registers and key archives to store k_{i+1}^{att}. In step 2d, $\mathit{AuthStages}(\mathcal{N}(D_S^{i+1}))$ implements checks to establish that next stage $\mathcal{N}(D_S^{i+1})$ can be trusted and, thus, should next receive control.

A stage $\mathcal{N}(D_S^{i+1})$ 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 in step 2d by a predicate *AuthStages*, which is assumed to be satisfied only by names of stages that will correctly execute the protocol. An implementation of *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.

The use of hardware-provided gating functions, key registers, and key archives is not the only way to ensure that an attestation private key k_{i+1}^{att} 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 $k_{i+1}^{\text{att}}\text{-}\mathbb{S}(\cdot)$. Once stage $\mathcal{N}(D_S^{i+1})$ starts executing, no predecessor will execute. So access restrictions that a gating function $[k_{i+1}^{\text{att}}\text{-}\mathbb{S}](\cdot)$ would impose can be achieved simply by $\mathcal{N}(D_S^{i+1})$ deleting k_{i+1}^{att} when $\mathcal{N}(D_S^{i+1})$ relinquishes control to its next stage.

A proof below shows that the certificates in $\mathit{AttCerts}_S$ suffice, as required, to establish K_i^{att} **speaksfor** $\mathcal{N}(D_S^i)$ for $0 \le i \le n$. That proof assumes certificates in $\mathit{AttCerts}_S$ are current rather than generated prior to the last reboot. The

1. $R \longrightarrow S$: $\langle$Att: $r\rangle$ for r a fresh nonce.
2. $S \longrightarrow R$: k^{id}_{hw}-$\mathbb{S}$(AttRply: $AttCerts_S$, D_S, v), where $v = k^{\text{id}}_{hw}$-$\mathbb{S}(r, \texttt{mr}_0)$
3. R: Accept remote attestation certificates in $AttCerts_S$ provided:
   (a) K^{id}_{hw} verifies digital signatures received step 2.
   (b) Nonce r received step 2 is same value as sent in step 1.
   (c) Using $AttCerts_S$, D_S, and v received step 2 check that:
   - $AttCerts_S$ implies K **speaksfor** S for some K.
   - D_S satisfies $S = \mathcal{N}(D_S)$,
   - D_S incorporates a descriptor giving a value for $\texttt{mr}_0$ that is consistent with v.

Figure 11.4: Defense Against Remote Attestation Replays

remote attestation protocol in Figure 11.2 defended against such replay attacks by having initiator R submit a fresh nonce r as a challenge. But that defense is not feasible for a protocol (e.g., Figure 11.3) that executes before remote attestation requests have been made.

One way to detect whether a certificate in $AttCerts_S$ is current, would be for the processor running S to have an instruction that computes response k^{id}_{hw}-$\mathbb{S}(r, \texttt{mr}_0)$ for any r, where register $\texttt{mr}_0$ can be read, cannot be written, and is incremented with each reboot.[13] A response is deemed current if it is produced when the value of $\texttt{mr}_0$ is the same as its value when the check is being made. So to check whether a response is current, it suffices to check whether (i) the response includes a value that is a Collision Resistant and Preimage Resistant function of $\texttt{mr}_0$ and (ii) that value is consistent with the current value of $\texttt{mr}_0$.

Responses in $AttCerts_S$ include $\mathcal{N}(D^i_S)$, which is a Collision Resistant and Preimage Resistant function of description D^i_S. So, provided the value of $\texttt{mr}_0$ is incorporated as a descriptor in each D^i_S then each element of $AttCerts_S$ does include the information needed for defending against replay attacks. Figure 11.4 gives such a protocol. It assumes initiator R knows a verification key K^{id}_{hw} for digital signatures produced by the processor hw purportedly executing S.

Note there is an alternative to requiring that the processor have a register $\texttt{mr}_0$ and an instruction to produce a certificate k^{id}_{hw}-$\mathbb{S}(r, \texttt{mr}_0)$. The protocol of Figure 11.4 also works if each description D^i_S includes a certificate signed by a remote trusted third party and containing challenge r along with a timestamp or sequence number that can be checked.

Formal Analysis of System Startup Attestation Protocol.* We prove that K^{att}_i **speaksfor $\mathcal{N}(D^i_S)$ for $0 \le i \le n$ can be inferred from $AttCerts_S$ after a system having description D_S has run System Startup Attestation Protocol of Figure 11.3. The proof is by induction on i.

[13] Instruction `KRgetCurConf` described on page 311 provides the required functionality.

The base case is to prove K_0^{att} **speaksfor** $\mathcal{N}(D_S^0)$. Step 1 of System Startup Attestation Protocol in Figure 11.3 sets $\mathcal{N}(D_S^0)$ to $\mathcal{N}(hw)$. By definition, $\mathcal{N}(hw) = K_{hw}^{\text{id}}$ holds. Since K_0^{att} **speaksfor** K_0^{att} is trivially valid, substitution of equals for equals yields K_0^{att} **speaksfor** $\mathcal{N}(D_S^0)$, as needed.

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

$$K_i^{\text{att}} \textbf{ says } K_{i+1}^{\text{att}} \textbf{ speaksfor } \mathcal{N}(D_S^{i+1}).$$

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

$$\mathcal{N}(D_S^i) \textbf{ says } K_{i+1}^{\text{att}} \textbf{ speaksfor } \mathcal{N}(D_S^{i+1}).$$

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

$$\mathcal{N}(D_S^i) \textbf{ speaksfor } \mathcal{N}(D_S^{i+1})$$

so CAL inference rule (9.16) derives

$$\mathcal{N}(D_S^{i+1}) \textbf{ says } K_{i+1}^{\text{att}} \textbf{ speaksfor } \mathcal{N}(D_S^{i+1}).$$

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

Trusted Boot and Secure Boot. In addition to their role in the remote attestation protocol of Figure 11.3, $AttCerts_S$ and *AuthStages* are useful for establishing trust in a local 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 local environment by checking whether $AttCerts_S$ 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 $AttCerts_S$. 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 $AttCerts_S$.[14] Provided a system startup stage that updates $AttCerts_S$ (step 2c of Figure 11.3) also updates `rt`, then reading `rt` suffices for checking whether $AttCerts_S$ is consistent with some predetermined set of certificates characterizing an execution environment that can be trusted. If the check passes, then the boot can be trusted.

[14] Thus, a measurement register (page 307) could serve as `rt`.

As with $\mathcal{N}(D_S)$, changes to D_S have unpredictable effects on the value in `rt`. That means installing a new 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 11.3. Step 2d of that protocol implies system startup will halt execution rather than transfer control to an unanticipated stage, thereby implementing a belief 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.

11.5 Other Uses

11.5.1 Full 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 and, 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 a disk block. Therefore, shared-key cryptographic routines implemented by software are the better choice. By using length-preserving shared-key encryption, block addresses on the disk and the disk layout itself do not have to be changed in order to accomodate storing encrypted disk blocks in place of plaintext disk blocks. However, with no space to incorporate redundant information, length-preserving encryption schemes cannot protect integrity. So disk encryption schemes typically protect the confidentiality, but not the integrity, of disk blocks.

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 time. 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[15] when a laptop is powered down or enters hibernation mode, the disk key must be stored by some non-volatile device. The disk is an obvious choice. To protect this stored copy of the disk key (since it cannot be encrypted using 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 a disk key that is not being kept 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 incorporating the routines to encrypt and decrypt disk blocks into the operating system, we avoid the need to include this functionality in each application that uses the disk. A design that put the encrypt and decrypt routines 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

[15] 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.

transform and move blocks between the encrypted and plaintext block caches. Note that boot blocks on the disk would not be encrypted, since those blocks are read from disk and executed before the operating system is available to perform decryption.

11.5.2 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.[16]

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 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 $\mathcal{N}(D_P) = P$ holds. Then the client retrieves the auxiliary information associated with description D_P, performs Checking Integrity of Auxiliary Information (page 305), 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 is required 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

[16] Clients, in turn, need to know the corresponding public keys for verification of quoted bit strings and for encryption that the unbinding key decrypts.

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.[17]

11.5.3 Digital Rights Management (DRM)

Computer networks offer an attractive infrastructure for distributing digital objects to customers. But certain digital objects require that access control 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 might need to authorize different operations (e.g., view, update, copy, print, or transfer) 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 that arise from converting between digital and analog formats. Active content (e.g., games and simulators), however, cannot be stolen in this way since the value is in having the capability to interact rather than having a record of the output. So active content can be monetized by enforcing restrictions on operations.

Support for measured principals enables hosts and applications that should be trusted to be distinguished from those that should not. That ability to discriminate, 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

[17] 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.

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 a locally generated 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: K_O encrypted using K_P and digital object O encrypted using K_O. P uses unbinding key k_P to recover K_O, which P uses to decrypt the digital object.

11.6 Possible Abuses and Benefits

Potential for Abuse. 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 applications can benefit from such control, it can be abused.

One abuse is *software vendor lock-in*[18] 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. 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.[19] A third exam-

[18] 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.

[19] As example of such a policy is Supreme Court Justice Potter Stewart's widely quoted

ple 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.

Benefits. Users also can benefit from 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.[20] 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 given to the principal identified with 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

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.

[20] 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.

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 §11.4.2 for remote attestation at system startup was first described there.

Fast forward a decade. The PC, 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[21] (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 *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 §11.5) 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.

In anticipation of NGSCB, a policy debate started (distilled in §11.5) about rights accompanying ownership of platforms and digital content, potential for 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 *trusted platform module* (TPM), would provide NGSCB with the needed hardware support for gating functions to provide sealing, quoting, and unbinding. The TPM specification was developed by the Trusted Computing Platform Alliance[22] (TCPA), formed in 1999 for this purpose by Compaq[23], HP, IBM, Intel, and Microsoft. (Other companies have since joined.) The material in §11.3 describes a simplified TPM.[24] See the Trusted

[21] The initial name was Palladium. Microsoft changed the name in January 2003.

[22] Trusted Computing Platform Alliance was renamed Trusted Computing Group (TCG) in April 2003.

[23] In June 1998, Digital Equipment Corporation, on the brink of insolvency, had been acquired by Compaq, a PC manufacturer.

[24] That discussion benefitted greatly from Ariel Segall's creative commons video course [27]

Computing Group's website [33] for the current, 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 [15] was among the first efforts, implementing a hypervisor that supports two kinds of virtual machines. An *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 *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. [5] extends Xen [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 [11]; Yap and Tomlinson [37] discuss para-virtualization for realizing a version 2.0 TPM.

CloudProxy [20, 32], which is intended for deployment in a cloud datacenter (and is the basis for the cloud-hosted services implementation sketched in §11.5), also offers a para-virtualization. The "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. [25], explores other operating system support for attestation of a software stack. In this work, a Linux kernel was modified to make and append *integrity measurements* onto *measurement lists* that are updated if an executable is loaded or if a (new) `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. [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 and/or other files. Since an executable must be analyzed to infer properties of its executions, the connection between a principal's name and its properties

about TPMs.

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 304 exemplify this. Sadeghi and Stüble [23] discusses having property descriptions be certificates from trusted third parties; their subsequent work with Winandy [24] avoids third-party dependence by delegating to analyzers that are part of the system.

The Nexus[25] [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 NetQuery [28] 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 restricted. 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.

With their Citidal architecture [36], IBM researchers circa 1991 turned their attention to security in distributed systems comprising devices located where

[25] Nexus also is the name that Microsoft gave to one of the NGSCB components.

attackers had physical access. Earlier IBM research efforts μABYSS[26] [34] and ABYSS [35] had explored obstructing physical access to hardware for cryptographic operations and storing keys. Building on this earlier 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[27] 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, building 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 cannot be described by a sequence of stages, because call/return transfers of control are used to execute initialization for various peripheral devices.

Bibliography

[1] Ross Anderson. Cryptography and competition policy: Issues with trusted computing. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, PODC 03, pages 3–10, New York, NY, USA, 2003. Association for Computing Machinery.

[2] Bill Arbaugh. Improving the TCPA specification. *Computer*, 35(8):77–79, August 2002.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on*

[26] ABYSS is an acronym for A Basic Yorktown Security System. The research group was located at IBM Research in Yorktown Heights, NY.

[27] This belief is no longer completely valid. Progress on homomorphic encryption means that certain computations with encrypted data now can be performed.

Operating Systems Principles, SOSP 03, pages 164–177, New York, NY, USA, Ocotber 2003. Association for Computing Machinery.

[4] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, pages 531–545, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[5] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the trusted platform module. In *Proceedings of the 15th Conference on USENIX Security Symposium – Volume 15*, USENIX-SS06, USA, 2006. USENIX Association.

[6] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[7] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, August 2016. USENIX Association.

[8] Dorothy E. Denning. Secure personal computing in an insecure network. *Communications of the ACM*, 22(8):476–482, August 1979.

[9] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A trusted open platform. *Computer*, 36(7):55–62, July 2003.

[10] Paul England and Butler W. Lampson. Secure execution of program code. U.S. Patent 6,651,171 B1. Filed April 1999, issued November 2003.

[11] Paul England and Jork Loeser. Para-virtualized TPM sharing. In *Proceedings of the 1st International Conference on Trusted Computing and Trust in Information Technologies: Trusted Computing – Challenges and Applications*, Trust 08, pages 119–132, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] Paul England and Marcus Peinado. Authenticated operation of open computing devices. In Lynn Margaret Batten and Jennifer Seberry, editors, *Information Security and Privacy, 7th Australian Conference, ACISP 2002, Melbourne, Australia, July 3-5, 2002, Proceedings*, volume 2384 of *Lecture Notes in Computer Science*, pages 346–361. Springer, 2002.

[13] David Farber, William Arbaugh, and Jonathan Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP 97, pages 65–71. IEEE Computer Society, September 1997.

[14] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from

software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP 17, pages 287–305, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP 03, pages 193–206. Association for Computing Machinery, 2003.

[16] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The Digital distributed system security architecture. In *Proceedings of 12th National Computer Security Conference*, pages 305–319. National Institute of Standards and Technology, National Computer Security Center, October 1989.

[17] Eric von Hippel. *Democratizing Innovation*. MIT Press, 2005.

[18] Intel Corporation. *Intel Architecture Instruction Set Extensions Programming Reference*, August 2015. Document number 319433–023.

[19] Stephen T. Kent, Karl Auerbach, Dorothy E. Denning, Doug Bates, and Ronald S. Lemos. Technical correspondence: On secure personal computing. *Communincations of the ACM*, 23(1):35–40, January 1980.

[20] John Manferdelli, Tom Roeder, and Fred Schneider. The CloudProxy Tao for trusted computing. Technical Report UCB/EECS-2013-135, EECS Department, University of California, Berkeley, July 2013.

[21] Chris Mitchell. *Trusted Computing*. Institution of Engineering and Technology, 2005.

[22] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. *Bootstrapping Trust in Modern Computers*. Springer Publishing Company, Incorporated, 2011.

[23] Ahmad-Reza Sadeghi and Christian Stüble. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *Proceedings of the 2004 Workshop on New Security Paradigms*, NSPW 04, pages 67–77, New York, NY, USA, 2004. Association for Computing Machinery.

[24] Ahmad-Reza Sadeghi, Christian Stüble, and Marcel Winandy. Property-based TPM virtualization. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *Information Security*, pages 1–16, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[25] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium – Volume 13*, SSYM04, page 16, USA, 2004. USENIX Association.

[26] Fred Schneider. Trusted computing in context. *IEEE Security and Privacy*, 5(2):45, 2007.

[27] Ariel Segall. Introduction to trusted computing. http://opensecuritytraining.info/IntroToTrustedComputing.html, January 2013. Accessed: January 15, 2020.

[28] Alan Shieh, Emin Gün Sirer, and Fred B. Schneider. NetQuery: A knowledge plane for reasoning about network properties. *SIGCOMM Comput. Commun. Rev.*, 41(4):278–289, August 2011.

[29] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP 11, pages 249–264. Association for Computing Machinery, 2011.

[30] Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(9):831–860, April 1999.

[31] Richard Stallman. Can you trust your computer? https://www.gnu.org/philosophy/can-you-trust.html, October 2002. Accessed: January 15, 2020.

[32] Wei Yang Tan, Rohit Sinha, John L. Manferdelli, and Sanjit A. Seshia. Formal modeling and verification of CloudProxy. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments*, pages 87–104. Springer International Publishing, 2014.

[33] Trusted Computing Group. TPM 1.2 main specification. https://trustedcomputinggroup.org/resource/tpm-main-specification/, October 2003. Accessed: January 15, 2020.

[34] S. H. Weingart. Physical security for the ABYSS system. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 52–52. IEEE Computer Society Press, April 1987.

[35] S. R. White and Liam Comerford. ABYSS: A trusted architecture for software protection. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 38–38. IEEE Computer Society Press, April 1987.

[36] Steve R. White, Steve H. Weingart, Willaim C. Arnold, and Elaine R. Palmer. Introduction to the Citadel architecutre: Security in physically exposed enviornments. Technical Report RC 16672 (#73914), IBM Corporation, March 1991.

[37] Jiun Yi Yap and Allan Tomlinso. Para-virtualizing the Trusted Platform Module: An enterprise framework based on version 2.0 specification. In

Proceedings of the 5th International Conference on Trusted Systems – Volume 8292, INTRUST 2013, pages 1–16, Berlin, Heidelberg, 2013. Springer-Verlag.