Authenticated Operation of Open Computing Devices

Paul England and Marcus Peinado

Microsoft
Redmond, WA 98052, USA
(pengland,marcuspe)@microsoft.com

Abstract. We describe how an open computing device can be extended to allow individual programs and operating systems to have exclusive access to cryptographic keys. This problem is of fundamental importance in areas such as virus protection, protection of servers from network attacks, network administration and copy protection. We seek a system that can be unconditionally robust against software attacks. This requires measures in hardware and in software. Our analysis allows us to minimize the amount of additional hardware needed to support the system.

1 Introduction

Consider a computing platform that allows arbitrary software to be executed. This paper investigates how individual operating-systems and programs can be given exclusive access to platform resources, even in the presence of adversarial software. We focus particularly on one type of resource: cryptographic keys. In this context, our goal is to allow individual programs to use or gain access to cryptographic keys, such that no other program can use or gain access to these keys. We call this mode of operation of a computing device *authenticated operation.*

Authenticated operation is of fundamental importance for a large range of applications of computer security, such as protecting personal data from viruses, protecting confidential server data from network attacks, network administration, copy protection, and trustworthy distributed computing. Authenticated operation allows different programs, which execute on the same computer without being in a particular trust relationship, to preserve their cryptographic resources irrespective of the actions of other software. This allows software to be compartmentalized and the size of the trusted computing base [1] to be reduced.

We work with the access control model [5]. In this model guards control access to resources. Guards can be implemented in hardware or software, and receive resource-access requests from principals. The guard can grant or deny access. Typically, the guard is a server, which is contacted by requesting clients over a network. In this case, the guard decides whether the request should be granted based on credentials (e.g. certificates, passwords) sent by the requesting principal over the network. Typical resources include files, printers etc.

Authenticated operation leads us to analyze the case in which the resource, the guard and the requesting principals are located in a single computing device. In our analysis, the requesting principals are programs, and the guard gates access to the resources (keys) based on the identity of the requesting program. It ascertains this identity by directly inspecting and controlling the program. While our descriptions apply to arbitrary resources, we focus on cryptographic keys. We can represent the different layers present in modern computers (e.g. hardware, bios, kernel, application software) by composing several instances of the components of the access control model. That is, a principal requesting access to resources in lower layers can also act as a guard to higher layers. In concrete terms, a (hardware) platform guard can control access to cryptographic resources based on the identity of the running operating system. Similarly, the operating system can grant selective cryptographic services to the application programs that it runs. Authenticated operation also allows program and operating-system authentication to be extended across networks. We will show how a network guard can authenticate software on hosts that may not be under its direct control.

The first part of this paper describes an architecture that enables authenticated operation. In general, programs that implement guards and the corresponding resources have to be isolated from other – potentially adversarial – programs executing on the same device. Furthermore, guards have to be able to authenticate requesting programs. Such authentication cannot be based on cryptographic protocols which require the requesting program to access cryptographic keys since passing the authentication step is a prerequisite to being granted access to the keys. We will describe methods for isolation, authentication and initialization of programs that work within this restriction.

The second part of the paper introduces several primitives, by means of which guards can expose restricted resources. We define the concepts of a *gating function*, *sealed storage* and *remote authentication* as general abstractions for cryptographic resource control. We show that the different gating functions we consider can be reduced to a set of three functions.

The third part of the paper describes the hardware extensions necessary to support authenticated operation. It was the goal of our analysis to identify a minimal set of hardware changes. This has the benefit of minimizing the cost of implementing these changes, in addition to identifying the true hardware primitives needed to enable authenticated operation.

Our attack model includes arbitrary software attacks. That is, given correct hardware and correct guard software, no program should be able to gain unauthorized access to resources – even in the presence of arbitrary adversarial software.

The definitions in this paper are abstract, so that they can apply to a broad class of computing devices. At the same time, it was our goal to show that our descriptions apply directly to real-world computers.

1.1 Related Work

This paper builds on several concepts of [10] and [7]. However, while [10] develops a general theory for authentication in distributed system, we focus on authenticated operation in a single stand-alone device.

Several authors describe systems that implement secure boot [3, 8] or secure coprocessors [15, 13, 12, 14]. While there are similarities between some of the techniques used there and authenticated operation, these works do not describe systems for authenticated operation. Secure boot focuses on the CPU as the only resource and achieves security by restricting the software that can execute on the device. Similarly, a secure coprocessor is an isolated processing environment, into which only authorized software is admitted. In contrast, authenticated operation imposes no restriction on admissible software. Instead, it allows each program exclusive access to a unique resource.

Secure booting approaches are typically based on a layered model, in which every layer i is isolated from all higher layers. Execution begins in a well-defined initial state of the bottom layer (hardware reset). Control is transferred successively from layer i to layer $i+1$ for $i \geq 1$. Before layer i transfers control to layer $i+1$, it authenticates the software in layer $i+1$. Secure boot implements an access control policy, which is typially PKI based. That is, code is expected to be accompanied by credentials in the form of public key certificates. In the simplest case, a file containing executable code can be loaded and executed in layer $i+1$ if and only if it is accompanied by a valid certificate chain, whose signatures are rooted in a public key known to the access control logic in layer i. This key constitutes the access control policy. If the credentials of a given binary file are in accordance with the access policy, the file is loaded and control is transferred to a well-defined entry point in the loaded image for layer $i+1$. Otherwise, layer i retains control and reports an error or tries to obtain software that is authorized to run in layer $i+1$ [3].

The specification of the Trusted Computing Platform Alliance (TCPA) [2] describes a hardware device that exposes several functions, which are related to authenticated operation. However, the overall description falls short of providing unconditional security against software attacks on a general purpose computing device. Furthermore, the hardware capabilities prescribed by TCPA are clearly not minimal.

The problem of allowing programs to maintain exclusive access to secrets is being studied in connection with software tamper-resistance or code obfuscation techniques (e.g. [4]). However, these software-only techniques are subject to attacks by software. In the absence of provable lower bounds for the robustness of software tamper resistance techniques and in light of complexity theoretic evidence against strong software tamper resistance [5], the true strength of these techniques is unclear. In contrast, this paper studies a minimal set of hardware measures, which allow the overall system to be unconditionally secure against attacks by software.

The rest of this paper is structured as follows. Section 2 describes how the access control model applies to authenticated operation and how isolation, authentication and initialization of guard components can be implemented. Section 3 defines several functions that can be used by guard components to expose access to resources and analyzes their relationship. Section 4 describes how existing computer hardware can be extended to enable authenticated operation. Finally, Sect. 5 concludes the paper.

2 System Architecture

This section identifies a small set of system capabilities, which are necessary to enable authenticated operation, and describes how these capabilities can be implemented. As these capabilities and implementations are quite general in nature, we describe them in a machine model which contains only those properties of real-world computers that are relevant to our discussion. This approach has the benefit of making the level of generality of our results obvious and of not burdening the discussion with unnecessary detail. However, we stress that our results are directly applicable to concrete real-world computers. This applicability is demonstrated in Sect. 4.

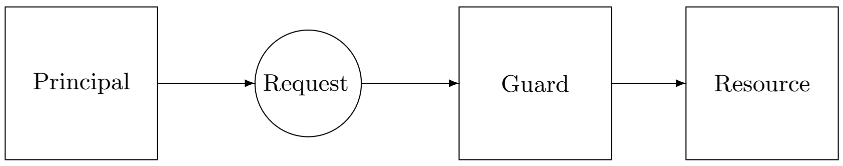

Fig. 1. Components of the access control model [9, 10].

2.1 System Model

Figure 1 displays the components of the access control model [9]. *Principals* make *requests* to access-protected *resources*. Each request is received by a *guard* – a component, which controls access to a resource. A guard examines requests and decides whether to grant each request based on an access policy for the resource and information, such as the identity of the principal that issued the request.

This paper focuses on the case in which the protected resources are cryptographic keys. In this context, we distinguish between *disclosure guards* and *service guards*. Service guards perform certain operations (encryption, signing etc.) with the resource (key) at the request of principals without disclosing the key. In contrast, disclosure guards reveal the protected key to authorized requestors.

As described in the introduction, we are interested in the case in which all principals and guards are programs executing on a single computing device. We assume that the computing device is a programmable, state-based computer with a central processing unit (CPU) and memory. The memory is used to store state information (data) and transition rules (programs). The CPU executes instructions stored at a memory location, which is identified by an internal register (instruction pointer IP). Finally, at least one protectable resource is embedded in the device. In the rest of the paper, we will refer to this semi-formal definition as a *computing device* or our *machine model.*

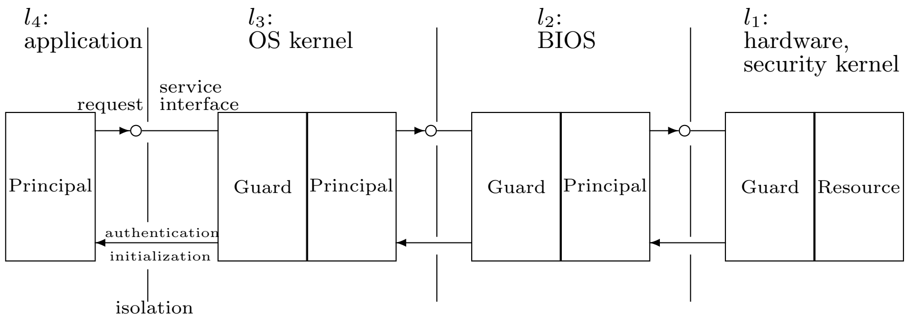

Fig. 2. A layered system in the access control model.

In practice, the principals on a computer can often be categorized into a small number n of hierarchical layers (e.g. hardware, operating system kernel, application software) $l_1, \ldots, l_n$. Figure 2 shows how $n > 2$ layers can by incorporated into the access control model by composing several instances of its components. In our model, the lowest layer guards a root resource. Programs in the intermediate layers act as principals that request access from the next lower layer. At the same time, they act as guards towards principals in the next layer. The significance of the intermediate layers in our model is twofold: (a) They correspond to properties of real-world computers. (b) The intermediate layers can add functionality for principals in higher layers. This is relevant if the guard in the lowest layer is simple (e.g. a cheap hardware implementation).

2.2 System Capabilities

We say that a program has *protected access* to a cryptographic resource K if no other program can gain the same access to K, with the exception of the guard of the resource. Clearly, the exception is necessary in the access control model, since the guard of a resource has at least as much access to the resource as a principal that requests access from the guard. Note that, in the layered model, the exception does not only apply to the guard g from which the principal is directly requesting access, but also to all guards, on which the access depends indirectly (Fig. 3).

Definition 1. *A computing device enables* authenticated operation *of software if it is possible to let every program obtain protected access (from a disclosure guard or from a service guard) to at least one cryptographic resource.*

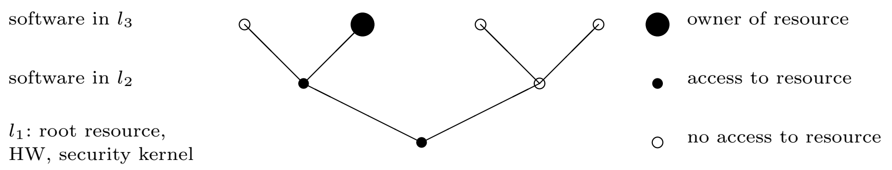

Fig. 3. Protected access in the access control model: Only the programs (guards) on the path from program A to the root can access the resource.

The rest of this section describes an architecture that enables authenticated operation.

Definition 2. *We say that a program C is* isolated *from another program D if (a) there is memory that can be accessed by C but not by D and (b) D cannot initiate execution of C, except, possibly, at a well-defined entry point (determined by C).*

A program is given by its transition rules (executable code) and by its initial state (entry point or initial value of the instruction pointer IP). The first condition of Def. 1 guarantees integrity of the program code and the state information of C, even in the presence of adversarial behavior by D, since these data can be stored in the memory that cannot be accessed by D. This condition also allows C to protect confidential data (e.g. cryptographic secrets) from observation by D. The second condition guarantees that D cannot subvert the behavior of C by choosing the initial state adversarially.

Definition 3. *Given a computing device, we say that a program C can* authenticate *a program D if C is able to identify the transition rules (program code) and the initial state of D.*

We require the computing device to enable isolation for any program C from any other program D, with the exception of a single program E_j for each layer $j < i$, where i is the layer of C (cf. Fig 3). Furthermore, for any layer i, the computing device has to enable a program executing in layer i to authenticate at least some programs in layer $i + 1$.

The first requirement protects programs from observation and interference by any program, except for the sequence $E_1, E_2, \ldots, E_{i-1}$ of guards through which C request access to its resources. As explained above, protection of C from these programs is not meaningful, since they have the same access to C's resources as C itself. The second requirement allows a program to act as a guard for requests from principals in the next layer. These two observations give rise to an inductive argument that programs in any layer can act as guards for resources by requesting access to a resource from their predecessor, protecting their integrity and the resource through isolation and authenticating requests from principals in the next layer. We summarize these observations in the following

Fact 1 *A computing device that enables authentication and isolation as described above, enables authenticated execution.*

Implementation: Isolation can be implemented by means of physical memory protections. We call this approach *isolation in space.* For example, the ring and virtual memory protections found in modern microprocessors are sufficient to implement isolation. An operating system kernel (layer i) running in privileged mode can set up page tables for applications (layer $i+1$), such that any application can only access those parts of physical memory that the operating system kernel chooses to map into the application's virtual address space. Furthermore, the kernel restricts applications' privileges so that they cannot change the memory mapping, and ensures that applications can initiate execution of kernel code only at well defined entry points (e.g. system calls).

A second approach to implementing isolation between two layers is to separate their execution in time. A program in a first layer i executes to completion, makes certain resources unavailable and terminates. Subsequently, control is transferred to the next layer $i+1$. We call this approach *isolation in time.*

Authentication occurs between subsequent layers ($j = i + 1$). C has to authenticate the program (transition rules) and the initial state of the configuration of j. The former can be achieved by letting C inspect the program in layer j. That is, typically C reads the memory, which contains the program for layer j and computes a cryptographic digest over this memory range.[1]

The second task for C is to identify the initial state of D. In general, it is not possible to determine the initial state of a program from its state at an arbitrary execution stage. More precisely, given a program P and two states σ_1 and σ_2, the question whether a computation of P could evolve σ_1 into σ_2 is undecidable. Thus, C has to control the initial state of D. In practical terms, this means that C can only ascertain the initial state σ_1 of D if C initiates the execution of D at σ_1.

In summary, in order to authenticate D, C inspects the memory contents it deems relevant (program and, possibly, data) and computes a cryptographic digest. After that, C transfers execution to a well-defined entry point of D.

3 Access Primitives

This section considers the case in which the resources are cryptographic keys. In this case, authenticated operation allows each operating system and application program to have exclusive access to a secret. The isolation requirement of the

[1] The goal at this point is only to ascertain the identity of the code, but not to evaluate statements made by other principals about the code. Thus, certificates are not necessary at this point.

previous section protects each secret from attacks by adversarial code. The authentication requirement allows the system to identify programs, such that each secret is disclosed only to the program that owns it.

We focus on two types of uses for secrets, which are bound to a program. The program can store long lived confidential and integrity protected information, such as banking records. Secondly, access to a secret allows a program to participate in cryptographic authentication protocols (for example with a remote server). We call these functions *sealed storage* and *remote authentication.* This section introduces several abstractions for the functions and investigates their relationship.

3.1 Gating Functions

Many of the abstractions we introduce in this section follow the same pattern. Given a request from a program, a guard establishes the identity of the program (i.e. authenticates it). If the program is not authorized to access or use the requested secret, the guard rejects the request. Otherwise, the guard computes some function of the secret and, possibly, further information provided by the program and returns the result. As an alternative to explicitly accepting or rejecting requests, the guard may always service the request, but bind the identity of the caller into the result. The latter approach is appropriate if the result returned by the guard does not contain confidential information (e.g. requests to use a secret to produce a digital signature). We use the term *gating functions* to refer to both types of functions. In either case, the guard has to authenticate the caller. In the rest of this section, we will model this step by a call to a function ID(), which returns a digest of the calling program.

3.2 Sealed Storage

The first class of gating functions we consider in this paper implements sealed storage. The purpose of sealed storage is to allow programs to store long lived secrets, such that only a well-defined set of programs (defined by the program that stores the secret) can retrieve them. In the simplest case, only the programs that originally stored (sealed) the secret can retrieve (unseal) it. Typically, the life time of these secrets will exceed the time of individual executions of the program. Isolation and a random number generator allows a configuration to maintain secrets during a single execution. Sealed storage allows a configuration to maintain secrets across different executions, which may not overlap in time. A layer l_i exposes sealed storage to the next layer l_{i+1} by means of the following interface.

Seal
Input: a secret s, the digest of a target program t
Output: c – an identifier for s
Description:
d =ID()

```
c = store $(s, t, d)$
return $c$
```

UnSeal
Input: c – an identifier for secrets
Output: a number s, the digest of a program t
Description:

```
$(s, t, d)$ = retrieve($c$)
if $t$ =ID() then return $(s, d)$
else fail
```

The Seal() operation stores its inputs (the secret and an indentifier for the configuration which may retrieve the secret) together with an identifier for the caller and returns an identifier, which allows the stored data to be referenced in subsequent UnSeal() operations. The function UnSeal() retrieves the data associated with its input, tests if the caller is authorized to read the secret (t =ID()) and returns the secret s and information that identifies its source (d). We note that Seal() and UnSeal() as well as the functions described later in this section can be easily extended to include more sophisticated access policies than the simple t =ID() equality check described here.

There are two approaches to implementing sealed storage. It is possible to implement store() and retrieve() by means of physically protected non-volatile memory – an expensive and limited resource. We prefer an implementation based on cryptography.

Cryptographic Implementations of Sealed Storage: This section describes an implementation, in which the required resource is a cryptographic key K, rather than physically protected memory. Store() will not physically store its inputs. Instead, it will produce a cryptographically protected output c, which contains its inputs in encrypted and integrity protected form. The former property results from applying a symmetric cipher to the input. The latter property results from applying a message authentication code MAC (e.g. the HMAC of [6]). This leads to the following implementation of Store() and Retrieve().

Store
Input: a bit string b
Output: a bit string c
Description:

```
$m = \mathrm{MAC}_{K_1}(b)$
$c = (m, \mathrm{Encrypt}_{K_2}(b))$
return $c$
```

Retrieve
Input: a bit string c
Output: a bit string b
Description:

```
Let $(m, d) = c$
$b =$Decrypt$_{K_2}(c))$
if $m = \text{MAC}_{K_1}(b)$ then return b
else fail
```

For technical reasons, we have partitioned K into two independent keys K_1 and K_2, in order to avoid using the same key for the MAC and the cipher. A number of alternatives for combining the MAC and the cipher exist. This type of implementation has the benefit of not imposing a limit on the number of secrets that can be stored – as is the case for physically protected memory. However, it should be observed that the purely cryptographic implementation has slightly weaker semantics. It guarantees only that corruption of c can be detected and that b cannot be retrieved from c without access to K_2. However, it it does not prevent corruption or the complete disappearance of c. This results in certain denial of service attacks, which are not possible for implementations based on physically protected memory.

3.3 Remote Authentication

The authentication mechanism described in Sect 2.2 allows a program to be authenticated to a closely coupled guard (a program on the same computing device). In particular, the authenticator must have direct read access to the memory containing the authenticated program code.

In this section, we introduce gating functions that allow programs to be authenticated even in the absence of a strong physical coupling to the authenticator (e.g. servers, smart cards). In this situation, authentication has to be based on cryptography. That is, both entities go through a cryptographic authentication protocol (cf. [11] for a summary of common protocols). This requires the authenticated configuration to have access to a secret, which, depending on the protocol, is typically a private key or a symmetric key. Going beyond pure cryptography, the computing device must tie the use of these authentication secrets to the identity of the configuration that requests their use – relying on the local authentication mechanism of Sect 2.2. Thus, the authenticator can not only establish the indentity of the computing device, but, more importantly, of the software stack executing on it.

The following two functions are the respective gating functions for public key signing and public key decryption. We assume that the guard implementing these functions has access to a private signing key K_s and a private decryption key K_d.

Quote
Input: an arbitrary data block a
Output: a signature s
Description:

```
$d =$ID()
return Signature$_{K_s}(d, a)$
```

PKUnseal
Input: a ciphertext block c
Output: a number s
Description:

```
$(d, s) = \mathrm{Decrypt}_{K_d}(c)$
if $d = \mathrm{ID}()$ then return $s$
else fail
```

The Quote operation returns a public key signature over the concatenation of its input and an identifier d for the calling program. The only assertion inherent in the signature is that it was performed at the request of d. Quote works in conjunction with a Verify() operation, which typically executes on a remote device, and which performs a standard public key signature verification and retrieves and evaluates d.

The PKUnseal operation is a version of public key decryption, which is gated on the identity of the caller. The result of the public key decryption of the input c is interpreted as a pair (d, s), where s is a secret and d identifies a configuration to which s may be revealed. If the caller of PKUnseal is not d then the operation fails. The input c is generated by a second operation *PKSeal*, which is typically executed on a remote device, and which performs a public key encryption of a pair (d, s).

Quote and PKUnseal are intended to be used in connection with public key authentication protocols [11]. Most protocols can be straightforwardly adapted, by replacing any call to public key decryption, public key encryption, signing, signature verification by a call to PKUnseal, PKSeal, Quote, Verify, respectively. Given appropriate management of program digests in PKSeal and Verify, the adapted protocols will prove to an authenticator which software is executing on a remote computing device.

3.4 Random Number Generation

The gating functions described so far are concerned with the restricted use of secrets that already exist on the computing device. It remains to address how these secrets can be initially obtained or generated. So far, our machine model is fully deterministic. Clearly, we have to provide a source of randomness. Typically, this source will be internal – implemented as a cryptographically strong random number generator in the machine hardware. We call the function through which the source of randomness is exposed *GetRandom()*. Let each call to GetRandom() return a fixed number of random bits.

Typically, calls to GetRandom() will be followed by calls to Seal(), in order to store the newly generated secret securely. We introduce the function GenSeal(), which combines both calls.

GenSeal
Input: the digests t_1, t_2 of two target programs
Output: c – an identifier for s

Description:

```
$d$ =ID()
$s$ =GenRandom()
c = store $(s, t_1, t_2, d)$
return $c$
```

GenSeal() is an optimization for certain restricted situations, which will be explained in Sect. 4.

3.5 Implementation of System Calls

The functions defined so far have the character of system calls. Given two subsequent layers i and $i+1$ of the computing device, software or hardware in layer i exposes system calls for sealed storage and remote authentication, which can be used from layer $i+1$. In general, the implementation of a system call mechanism depends strongly on the isolation mechanism between the two layers.

Isolation in space: In practice, the memory protections that enable isolation in space typically include provisions for system calls. The software in the calling layer $(i+1)$ issues a special instruction, which blocks its execution and passes control to a well-defined entry point of layer i. The software (or hardware) in layer i uses its secret to service the system call and returns control to the caller in layer $i+1$. In practice, the performance overhead is typically small.

Isolation in time: System calls across layers, which are isolated in time are more complicated and costly due to the fact that, by the time the system call is made from layer $i+1$, layer i is no longer active. In particular, its secrets have been made inaccessible. Servicing a system call which depends on these secrets requires reinitialization of layer i and, possibly, the layers below it. In the most extreme – and probably typical – case, servicing the system call requires a full reset of the device.

In light of the high cost of system calls under isolation in time, the following optimization is intended to improve system performance by replacing most system calls by library calls. Conceptually, any service guard functionality in layer i is moved into layer $i+1$ and converted into library code. Layer i acts only as a disclosure guard. After an initial GenSeal() or UnSeal() operation by the disclosure guard in layer i for the library code in layer $i+1$, the latter can cache the returned secret and expose sealed storage and remote authentication in a self contained way. That is, after the initial GenSeal() or UnSeal() operation by layer i, which can be executed during boot, all calls by code in layer $i+1$ to the functions defined in this section can be executed with minimal overhead as library calls within layer $i+1$ and without any further system calls into layer i.

We note, however, that this optimization results in somewhat weaker semantics for certain operations in the presence of security failures. For example, in any system that exposes Quote() by means of a service guard in layer i, security bugs in layer $i+1$ may allow an adversary to obtain false signatures by making system calls into Quote() *only until* the bug is repaired. In contrast, under the optimization, the signing key used by Quote() is available in layer $i + 1$. Thus, a security bug in layer $i + 1$ may allow an adversary to extract the signing key and to produce signatures even after the error has been corrected.

One consequence of the optimization described above is that the remote authentication functions (Quote(), PKUnseal()) can be implemented given only sealed storage and appropriate initialization. During initialization (e.g. manufacture of the device), a public key pair is generated, the private key is sealed to an appropriate software configuration, and the public key is certified. This observation has practical significance, as it allows devices without hardware or microcode support for public key cryptography to support remote authentication.

We conclude that a device with the following hardware primitives can implement sealed storage and remote authentication: (a) GenSeal(), (b) Unseal(), (c) an isolation mechanism, and (d) a means for correctly initializing the device. The next section will outline hardware implementations of these primitives.

4 Hardware Implementation

In this section we describe a family of hardware implementations that will enable platforms to support authenticated operation. As with higher layers in the system, the characteristics of the lowest layer (l_1) are a) secret key resources, b) privileged code that has access to these keys, and c) controlled initialization of the layer.

4.1 Layer 1: Initialization and Resources

Authenticated operation provides a strong binding between programs and secret keys. At higher layers, we have assumed that guards in lower layers guarantee this binding. At the lowest layer we do not have an underlying software guard that can gate access to the platform secrets; hence we need another mechanism to support the strong association of the l_1-keys to the l_1-program. The most straightforward way of accomplishing this binding is to require l_1 software be platform microcode or firmware that is not changeable following manufacture, and give the l_1 software unrestricted access to the l_1 keys. In the remainder of this paper we will call the platform-microcode the security kernel, and the l_1 keys the platform keys.

This mechanism is a simple form of secure boot. The platform will only pass control to a predetermined security kernel. The hardware behavior can also be explained as a simple resource guard that discloses the platform keys to the predefined security kernel.

We have no specific requirements regarding whether the platform keys and security-kernel firmware are part of the processor or in other platform components. In general, keys and code that are embedded into the microprocessor chip will be harder to subvert, but other manufacturing considerations can make off-chip solutions attractive.

Authenticated operation requires that programs are started in a controlled initial state. At higher levels, the software running at lower-levels can be entrusted to start execution at the correct entry point. At l_1 we need hardware to perform this function. Fortunately, on power-up or following reset, all processors already begin execution by following some well-defined deterministic sequence. In the simplest case the processor starts fetching and executing code from an architecturally-defined memory location. In this case (and with minor variations for the more complicated startup sequences), it is sufficient for hardware to ensure that the security kernel is the code that executes on startup.

Another requirement is that no other platform state can subvert execution of the security kernel. Reset and power-up provide a robust and a well-debugged state-clear for the processor. We will call the platform state change that is used to start or invoke the security kernel a security reset.

Finally, a device manufacturer must arrange for the generation or installation of the platform keys used by the l_1 implementation of Seal and Unseal. If the device is to be recognized as part of a PKI, the manufacturer must also certify a public key for the platform. This can be a platform-key used directly be l_1, or a key used by a higher layer (cf. Sect. 3.5).

Key generation and certification can be the responsibility of the CPU manufacturer, the responsibility of the OEM that assembles the CPU into a device, or both parties.

4.2 Layer 1: Isolation and Guard

Once the security kernel is executing it can use the isolation mechanisms we have described to protect itself from code executing at higher layers.

No additional platform support is needed to implement space isolation on most processors: an exiting privilege mode or level will suffice (as long as the hardware resource that allows access to the platform key can be protected from higher layers). However, to support time-isolation, we need hardware assistance to allow the security kernel to conceal the platform key before passing control to higher layers [10].

The simplest way to provide platform-key security in the time-isolation model is to employ a stateful guard circuit that we call a reset latch. A reset latch is a hardware circuit that has the property that it is open following reset or power-up, but any software at any time can programmatically close the latch [14]. Once closed, the latch remains closed unto the next reset or power-up. A platform that implements a time-isolated security-kernel should gate platform-key access on the state of a reset-latch, and the security-kernel should close the latch before passing control to higher layers. As we have already described, the security kernel will have to take additional actions like clearing memory and registers before passing control, but these steps are identical to those required at higher levels.

4.3 Layer 1: Service Invocation

If the platform employs space-isolation the security kernel must use privilege modes to protect itself and its platform keys from programs (e.g. operating systems) that it hosts. Furthermore, the security kernel must establish a system call interface for invocation of the authentication operations.

If the platform employs time-isolation, then the platform must also contain storage that survives a security-reset to pass parameters to service routines. To invoke a service, an operating system must prepare a command and parameter block in a memory location known to the security kernel and perform a security-reset. If the OS wishes to continue execution following the service call (as opposed to a simple restart) then it and the security kernel must take extra measures to ensure that this can be done reliably and safely.

To conclude, a security kernel that implements time isolation with a reset-latch can be built with existing processors with a very small amount of external or internal logic. Furthermore, if the primitives exposed are Unseal and GenSeal then the security kernel is very small.

5 Conclusions

We have defined authenticated operation and identified properties, which allow a computing device to enable authenticated operation. In particular, we have identified isolation and local authentication as critical capabilities and outlined implementation options for them. A critical component in the implementation is a booting procedure, which successively tranfers control from layer to layer through well-defined entry points.

The main goal of authenticated operation is to enable mutually distrustful software components on a computer to have exclusive access to cryptographic keys. We have discussed abstractions for the main uses of these keys (sealed storage and remote authentication), and we have tried to identify a minimal set of primitives, which enable these uses. Finally, we have outlined the hardware measures necessary to implement these primitives.

References

1. DOD 5200.28-STD. Department of defense trusted computer system evaluation criteria. December 1985.
2. Trusted Computing Platform Alliance. TCPA main specification version 1.1. http://www.trustedpc.org, 2001.
3. W. A. Arbaugh, D. J. Faber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 65–71, 1997.
4. D. Aucsmith. Tamper-resistant software: An implementation. In Ross Anderson, editor, *Information hiding: first international workshop, Cambridge, U.K.*, volume 1174 of *Lecture Notes in Computer Science*, pages 317–333. Springer-Verlag, 1996.

5. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Advances in Cryptology – CRYPTO 2001*, pages 1–18. Springer-Verlag, 2001.
6. M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology – Crypto'96*, number 1109 in Lecture Notes in CS, 1996.
7. P. England, J. DeTreville, and B. Lampson. A trusted open platform. Unpublished.
8. N. Itoi, W. A. Arbaugh, S. J. Pollack, and D. M. Reeves. Personal secure booting. In V. Varadharajan and Y. Mu, editors, *Information Security and Privacy – 6th Australasian Conference, ACISP 2001*. Springer-Verlag, 2001.
9. B. Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24, 1974.
10. B. Lampson, M. Abadi, and M. Burrows. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10:265–310, November 1992.
11. A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.
12. S. W. Smith and V. Austel. Trusting trusted hardware: Towards a formal model for programmable secure coprocessors. In *Proceedings of the Third USENIX Workshop on Electronic Commerce*, pages 83–98, 1998.
13. S. W. Smith, E. R. Palmer, and S. Weingart. Using a high-performance, programmable secure coprocessor. In *Proceedings of the Second International Conference on Financial Cryptography.* Springer-Verlag, 1998.
14. S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(8):831–860, April 1999.
15. B. Yee. *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon University, 1994.