A Comparison of CDSA to Cryptoki

Ruth Taylor (rct@epoch.ncsc.mil), National Security Agency
February 16, 1999

Keywords: CDSA, Cryptoki, CAPI, port

Abstract

The Common Data Security Architecture (CDSA) is a general security service architecture which has been standardized by the Open Group. This paper compares the CDSA CAPI to another well known low-level CAPI, RSA's PKCS #11 (Cryptoki).

Both CDSA and Cryptoki are low-level interfaces which satisfy criteria established by the NSA's CAPI Team. However, CDSA provides a security services infrastructure to several categories of security services, and therefore provides more auxiliary services to manage this more complex architecture. Additionally, Cryptoki provides a more direct interface to hardware cryptographic tokens. This paper maps calls in the APIs, describes differences between the two and how these may be handled, and considers porting issues.

1 Introduction

What Is a CAPI and Why Is It Useful?

A Cryptographic Application Programmer Interface, or CAPI, is a set of calls which allow an application programmer to access cryptographic functionality. A high-level CAPI allows a programmer with little cryptographic knowledge to request an operation (e.g. encrypt) without requiring the programmer to supply details on how the operation will be performed, whereas a low-level CAPI allows a more cryptographically-aware programmer to specify cryptographic details (e.g. algorithm and mode) by passing them as parameters to the CAPI call. The NSA CAPI Team has established CAPI evaluation criteria, and recommended a suite of CAPIs [3].

CAPIs are advantageous for several reasons. First, a correct, readily-available cryptographic library may be used by any application which conforms to the CAPI. Second, cryptographic software or hardware may be modified without changing any application code. Third, placing cryptographic code in a separate library decreases the likelihood that application bugs will illegally access sensitive cryptographic data (e.g. keys) or cryptographic code.[1] Even stronger protection is provided if the application and cryptographic code are run by separate processes. The strongest protection guarantees are provided if the system uses a secure operating system, as described in "Operating System Importance" on page 3.

General Introduction: CDSA and Cryptoki

The CSSM API and Cryptoki both define low-level interfaces to cryptographic functionality; they require applications to specify cryptographic algorithms and attributes when requesting a cryptographic operation. However, these CAPIs have several differences.

First, the CDSA specification defines interfaces to cryptographic, data storage, certificate, and trust policy libraries. In contrast, Cryptoki defines only an interface to cryptographic functionality. Therefore, CDSA is necessarily richer in auxiliary services (like module management), because more auxiliary services are needed to manage its more complex architecture.

Second, CDSA's CSSM API is designed to use either software or hardware Cryptographic Service Providers (CSPs)[2]. However, Cryptoki was originally designed to directly interface with a hardware cryptographic token (although it can interface to a software cryptographic module as well), and allows the application programmer more direct interaction with hardware tokens. "API Differences" on page 7 explores these differences further.

Introduction to CDSA

The CDSA specification was initially defined by Intel Architecture Labs, and has received the support of many influential companies including IBM, Netscape, TIS, Motorola, Sun, and Hewlett-Packard [4]. In October 1997, the CSSM API was added to the CAPI suite recommended by the NSA Cross Organizational CAPI Team [3], and CDSA was recommended for use in the Defense Advanced Research Projects Agency's (DARPA's) Advanced Information Technology Services Reference Architecture (AITS RA)[2]. In January 1998, CDSA was adopted as a commercial standard by the Open Group.

A complete description of the Common Data Security Architecture (CDSA) is given in [1]. CDSA provides a layered infrastructure which allows applications to access security functionality, and allows system administrators to

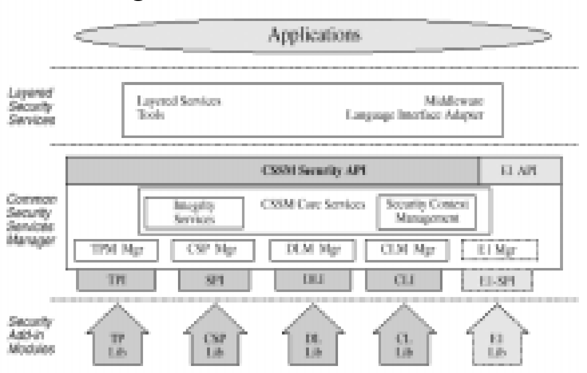
1.Vulnerabilities still exist if the cryptographic code is in a separate library. E.g., the Cryptoki `C_GetFunctionList` call returns function pointers which point directly to memory in the cryptographic library. The Cryptoki specification states that this memory should not be written to, but modification is possible if the library is writable. [5]

2."CSP" refers to a cryptographic module implemented by any vendor and plugged into CDSA. This should not be confused with Microsoft's use of "CSP" as a module which provides cryptographic services for the Microsoft Cryptographic API package.

"add-in" modules of their choice which implement the security functionality. The Common Security Services Manager (CSSM) manages the add-in modules. The introduction below progresses downward, describing first the application programmer interfaces (APIs), then the CSSM, and finally the service provider interfaces. Figure 1, extracted from [1], shows the CDSA architecture.

The CSSM API Specification [1] defines interfaces by which application programmers can access Cryptographic Service Provider (CSP) libraries, trust policy (TP) libraries, certificate libraries (CL), and data storage libraries (DL). CDSA also defines an optional key recovery API. Applications can access calls not defined by the CSSM API, but implemented by a service provider, via the `CSSM_PassThrough` mechanism.

Figure 1: CDSA Architecture



The CSSM manages add-in modules. It verifies the authenticity and integrity of service module sources before adding them to the CDSA environment, maintains a registry of the current add-in modules and their capabilities, caches user security contexts (which contain parameters to calls and possibly sensitive information), and responds to application queries concerning the availability and functionality of service provider modules. System administrators install and uninstall the security service modules using the CSSM Module Management Functions.

The CSSM includes Cryptographic, Trust Policy, Certificate, and Data Storage Service Managers, which map application service request calls to the lower level service provider calls, thus providing applications with service. These managers perform built-in security checks as well. Additional "Elective Module Managers" can be created to dynamically extend the system by adding new types of services, which are also managed by the CSSM.

The add-in CSPs, certificate libraries, trust libraries, data storage libraries, or elective add-in libraries must conform to the service provider interfaces which are lower-level than the CSSM API. CDSA's Service Provider Interface (SPI) for CSPs is defined in the CSSM SPI Specification [1]. CSPs can be implemented in hardware or software, and perform cryptographic operations like encryption, decryption, digital signing, key generation, random number generation, or computing message digests for data. CSPs plug into the Cryptographic Services Manager. CSPs can optionally support login/logout capability and privileged operations for CSP administrators. Because the model for administration can vary widely among CSPs, any such functions can be provided as PassThrough functions and are not part of the normal interface.

Applications using the CSSM API may either preallocate output memory buffers, or request that the CSP allocate this memory (in application memory-space) for the application. In the former case, an API call (`CSSM_QuerySize`) allows the application to obtain the necessary size for an output buffer given a particular function call; the application can then allocate this space and call the function. In the latter case, when the application establishes a session with the CSP (using the API call `CSSM_ModuleAttach`), it passes the CSP a table of basic memory management function pointers, and a heap pointer for the memory-space of the application. The CSP will then allocate memory in the application memory-space, but the application is responsible for freeing this memory. (The CSSM API includes the calls `CSSM_FreeInfo`, `CSSM_Freelist`, `CSSM_Free`, `CSSM_GetAPIMemoryFunctions`, `CSSM_FreeKey`, `CSSM_Free*Context`[3], and `CSSM_FreeModuleInfo` for memory deallocation.)

Introduction to Cryptoki

RSA's Public Key Cryptography Standard (PKCS) #11, also known as Cryptoki, is described in the PKCS #11 Cryptographic Token Interface Standard, Version 2.01 [5]. NSA's Cross Organizational CAPI Team recommended Cryptoki to meet the present and future needs of NSA in October 1997 [3]. Cryptoki is used in products such as Netscape Navigator. Cryptoki is an API to cryptography devices, and defines a single interface which applications and cryptomodules must conform to. Because it is simpler than CDSA, it does not incorporate as many auxiliary services to manage its infrastructure as CDSA.

Cryptoki has its own terminology. A "mechanism" is a cryptographic algorithm; a "token" is a module which uses a mechanism to perform cryptographic functions; and a "slot" is an abstract adaptor which holds a token. (The Cryptoki specification defines a basic set of mechanisms, although a compliant Cryptoki implementation is not required to support all of these.) A "session" is a logical connection between an application and a token.

3.`CSSM_Free*Context` refers to the family of calls where '*' equals `MAC, Signature, KeyGen`...

Tokens can support one or more sessions. Cryptoki mechanisms include many newly added official algorithms in NSA's MISSI suite. Token vendors can also define their own mechanisms for use with Cryptoki, but for interoperability, registration with RSA's PKCS is preferable. Tokens may define objects (defined below) and functions as "public", so that any user may access them, or "private", so that only authenticated, logged-in users may access them.

An application using Cryptoki must perform its own memory management. After making a Cryptoki call to determine the required output buffer size, the application must allocate this space before calling the function. Note that this is different from CDSA, which provides the application with the option of having memory allocated (in the application memory-space) by the cryptomodule.

CSSM API Contexts Vs. Cryptoki Objects

CSSM API contexts contain all the information needed to perform a cryptographic operation, like context type (e.g. key exchange, signature, or digest), algorithm (e.g. RSA, KEA, or MD5), and applicable attributes (e.g. key data, initialization vector, algorithm mode, padding, dates of validity for object, or passphrase). A context handle is passed as a parameter in the algorithm-independent CSSM CAPI calls. The CSSM API defines cryptographic algorithm identifiers which may be implemented by an add-in cryptographic module.

Cryptoki objects contain secret keys, public and private keys, application-defined data, or certificates. Objects are specific to algorithm and function (i.e. Cryptoki defines a specific structure for a DES3 secret key object), and their definitions in the Cryptoki specification describe the exact data format and information needed to represent that object. Attributes indicate which operation the objects are to be used for (encrypt, decrypt, sign, verify, wrap, unwrap, or derive). An object may belong to a token, in which case it may persist from session to session. Or, the object may be created for a session, and then destroyed upon closing of the session.

Cryptoki programmers set a "mechanism" variable to the algorithm they wish to use (e.g. DES); this variable is then passed as a parameter in the algorithm-independent Cryptoki calls. When needed, a mechanism parameter is used to pass additional data (like an Initialization Vector (IV) for an encryption algorithm).

Cryptoki objects and mechanisms together contain the same information held by a CDSA context. Both CSSM API context handles and Cryptoki object handles are passed as parameters to a cryptographic call in the CAPI, and specify details about the requested cryptographic operation. Both the CDSA and Cryptoki cryptographic libraries create the context or object, and return only a handle to the application, so that the application cannot directly manipulate the object or context.

2 Comments on CDSA and Cryptoki

Operating System Importance

Clearly, it is important to ensure that cryptomodules and infrastructure components (e.g. CDSA's CSSM) are written correctly and securely. For example, it must be verified that sensitive information is sufficiently protected (e.g. that private keys are tagged as "always sensitive", or that CSSM Module Managers do not share state information which results in confidentiality breaches). Additionally, an application programmer using CDSA or Cryptoki must be sufficiently knowledgeable of security to use the correct services and configurations to meet his security requirements.

Even if the above concerns are addressed, a cryptographic system may still be vulnerable if the operating system upon which it is running is not secure. [6][7] A trusted operating system can provide user level code with confidentiality, integrity, authentication, and assured delivery of the inter-process communication (IPC), and confidentiality and integrity for memory and long-term storage. (Though many examples below relate to CDSA, Cryptoki may use either hardware or software tokens, and therefore depends on the operating system in the same manner as CDSA.)

Secure IPC can provide confidentiality and integrity of data passed via IPC, identification and authentication (I&A) of the sender and receiver of IPC, and guaranteed delivery and invocation. Vulnerabilities may result if the cryptographic library and the application are in separate address spaces, and the system's IPC is insecure. For example, misuse or spoofing of cryptographic components can occur; a hardware cryptomodule which securely creates and stores keys, as well as a software cryptomodule, may be subject to unauthorized use. Or, the CSSM or cryptomodule could be spoofed to its caller, and the application can be spoofed to the CSSM or cryptomodule.

The possibility of spoofing is reduced with CDSA, because the CSSM must provide signed manifest credentials to applications. An application may use these credentials to authenticate the CSSM using EISL functions. Additionally, if the application itself has a signed manifest credential, it may use an EISL library to perform a self check, and the CSSM may check this application. However, there is a trade-off between threat mitigation (frequency of EISL checks) and performance costs. In Cryptoki, the legitimacy of a token may be established by authentication using certificates. However, if the Cryptoki cryptomodule is in user space, its code and certificates may be modified, weakening the confidence in the authentication.

Additionally, data passed via IPC could be subject to unauthorized interception and modification. For example, key data (raw or wrapped), pointers to key data, and passphrases for key access, may be passed via IPC when building a CDSA context, or when requesting an operation. This data could be copied and reused, or modified by malicious parties. (Both CDSA and Cryptoki allow keys to be protected from this threat by identifying them as sensitive or unextractable from the cryptomodule.) Or, a passphrase controlling cryptomodule access passed via insecure IPC may be obtained by applications or other devices monitoring the communication lines. [5] Plaintext could be intercepted and copied, or modified; and ciphertext could be modified, so that the corresponding plaintext cannot be retrieved from it. Cryptographic operation requests and contexts could be modified or deleted in route to the CSSM or cryptomodule. And, if the CSSM Module Managers execute in different address spaces, then messages sharing internal state would be sent via IPC, and these could be eavesdropped upon, modified, or deleted.

Finally, applications and supporting infrastructure components depend upon the OS to prevent unauthorized access to data and executables in memory or long term storage. In CDSA, malicious code or data modification can be detected, but not prevented, by EISL checks. These integrity checks may be performed at any time (e.g. before each code execution or data access). However, there is a trade-off between threat mitigation and performance costs. (More frequent EISL checks decrease the likelihood that maliciously modified code will be executed before the modifications are detected.)

CDSA assumes that keys and other sensitive security context information will be protected either by the CSP (e.g. if the keys are “never extractable” from the CSP), or by the application (e.g. if the keys are raw and extractable from the CSP). However, both applications and software CSPs run in user space, and rely on the operating system to protect this data from unauthorized reading, writing, or execution. Similarly, the software cryptographic modules could be modified by unauthorized parties, and this code could be used before the changes are detected by the EISL. The keys used to verify signatures of the EISL or Integrity Verification Kernels (IVKs, which verify the integrity of an individual software module) could be modified. And, the application calls to CDSA could be modified or deleted by malicious agents.[6]

Comparison Against NSA CAPI Criteria

The NSA CAPI Team established several criteria for evaluating CAPIs. These include algorithm independence (the CAPI allows an application programmer to specify a wide range of cryptographic algorithms); application independence (the CAPI offers services needed by a wide variety of applications); cryptomodule independence (the CAPI refers to a wide variety of cryptomodules); MISSI support; modularity and auxiliary services; safe programming and degree of cryptographic awareness (the CAPI uses consistent naming conventions, minimizes complexity of language features to prevent unintentional programming errors, and minimizes the amount of cryptographic expertise required of the application programmer); and security perimeter (the CAPI controls access to sensitive data, and does not allow movement of sensitive data beyond the security perimeter).[3]

Both the CSSM API and Cryptoki were found to be algorithm independent. Calls in both APIs specify the type of operation (e.g. encrypt, sign, verify) rather than the specific algorithm to be used (e.g. DES, RSA), and both CAPIs allow a system administrator or user to add software or hardware modules which implement the desired cryptographic algorithms. [1][3][5]

Similarly, both the CSSM API and Cryptoki were found to be application independent, because they offer a low-level interface to cryptographic operations, which can be used by different applications. Both were found to be cryptomodule independent; Cryptoki uses the token concept to abstract the cryptomodule, and CDSA uses the CSP to abstract the cryptomodule, and contexts to abstract module-specific data.

The NSA CAPI Team found that both Cryptoki and CDSA provide sufficient MISSI support. [3] The CSSM SPI lists over seventy algorithms which may be implemented by add-in cryptographic service providers, including KEA (MISSI’s Key Exchange Algorithm), BATON, JUNIPER, and SKIPJACK (all MISSI block ciphers). In addition, the CSSM API is extensible; any functionality provided by the module but not in the CSSM SPI is accessible to applications as a “PassThrough” function call. [1] Similarly, Cryptoki’s Version 2.01 defines key objects and mechanisms for KEA, BATON, JUNIPER, and SKIPJACK (all MISSI algorithms). Additionally, token vendors using Cryptoki may define their own mechanisms, but for inter-operability, registration with PKCS is preferable. [5]

The NSA CAPI Team found that CDSA was modular and provided sufficient auxiliary services.[3] CDSA is very modular by design; all cryptographic, trust policy, data storage, or certificate service modules are loaded as separate, add-in modules. The CSSM is composed of the Cryptographic, Trust Policy, Data Store, or Certificate Services Managers. Auxiliary services provided by the CSSM include dynamic module installation, attachment, and detachment; maintenance of a registry of the current modules and their capabilities (this registry may be queried by applications); module integrity checks; memory management; cryptographic context management; key generation; login/logout capability; password changing capability; callback capability; unique ID generation; and a tamperproof counter. A "callback" function can be defined when attaching a CSP; this function is executed when a predefined event occurs. Additionally, the CSSM checks the current CSSM and CSP versions against the version needed by applications.

Cryptoki also generally satisfies the modularity and auxiliary services criteria. [3] Auxiliary services provided by Cryptoki include login/logout capability, callback capability, key generation, and random data generation. Although Cryptoki does not provide explicit calls for cryptomodule verification, token authenticity can be achieved by distributing the token with a built-in, certified private/public key pair, by which the token can prove its identity. Users can obtain information on available mechanisms, objects, and slots. [5]

The NSA CAPI Team found that both CDSA and Cryptoki required a cryptographically aware programmer, but both were rated as requiring fairly "unsafe" programming. [3] CDSA and Cryptoki both use consistent naming conventions. CDSA precedes all SPI functions with the "`CSP_`" prefix, and all API functions with the "`CSSM_`" prefix. Aside from the prefixes, functions from the API and SPI with similar functionality have equivalent names. Cryptoki uses unique prefixes to distinguish data types, objects, attributes, functions, return values, and other unique features. Cryptoki objects are always initialized to default values (which may be modified at creation time), and always contain a set of required attributes.

With respect to complexity of language features, CDSA describes its intended audiences as experienced security and software architects, advanced programmers, and sophisticated users, who are familiar with network operating systems and high-end cryptography, and familiar with the basic capabilities and features of the protocols they are considering. CDSA requires advanced knowledge of the cryptographic algorithms to be used. Similarly, Cryptoki uses C in an advanced, object-oriented way, requiring advanced C programming skills. It requires in-depth knowledge of the algorithms used, and of the underlying token. Both require the programmer to know the correct sequence in which cryptographic calls should be made. [1][3][5]

The NSA CAPI Team found that both CDSA and Cryptoki sufficiently enforce the security perimeter. [3] CDSA provides several features which may enforce the security perimeter. First, key data generated by the `CSSM_GenerateKey` and `CSSM_GenerateKeyPair` calls can be wrapped and/or encoded, or a reference to a key. They may be labeled permanent or modifiable; extractable or non-extractable from the CSP; and private, sensitive or always sensitive.

Second, the CDSA application can only obtain a context through calls which return a handle to the context. (A context may contain sensitive information like pointers to key structures, which are in turn obtained from the `CSSM_GenerateKey` and `CSSM_GenerateKeyPair` calls.) Once the context has been created, the application can only identify the context and keys through the context handle (which is passed to the CSSM when requesting cryptographic operations).

Third, the CSP is responsible for secure storage of private keys; the `CSSM_CSP_CreateSignatureContext`, `CSSM_CSP_CreateAsymmetricContext`, and `CSSM_CSP_CreateKeyGenContext` functions all require a "passphrase" parameter to unlock private keys, and the `CSSM_CSP_CreateDeriveKeyContext` call requires a passphrase for signature operations. The CSP may optionally be responsible for storage of other objects, like certificates. Persistent storage can be implemented using a data storage library module, or be implemented within the CSP. Lastly, the CSSM environment is protected through module source verification using certificates, and module code integrity checks using a signed hash.

Cryptoki provides features which may enforce the security perimeter as well. First, a token can define private objects and functions, which can only be accessed after an authenticated user login. (A token may also define public objects and functions, which may be accessed without login.) Second, additional protection can be given to private or secret keys by marking them as "sensitive" or "unextractable". Sensitive keys must be wrapped if they are exported from the token, and unextractable keys may not be exported from the token. The Cryptoki Specification [5] notes that if protected memory is not available to store sensitive objects, then they may be encrypted using some derivation of a user's PIN; but this PIN may itself be compromised through weaknesses in the operating system IPC channels.

3 Mapping the Calls

This mapping considers all calls from the CSSM SPI and Cryptoki drafts, but only calls from the Core Services API and Cryptographic Services API from the CSSM API. For a complete mapping between calls in these interfaces, please see [9].

Generally, while CDSA and Cryptoki's cryptographic operation calls map one-to-one, other types of calls have one-to-many mappings from CDSA to Cryptoki, because the CSSM API is a slightly higher-level interface than Cryptoki. For example, preparing for a cryptographic operation in CDSA requires one context creation call, while preparing for this operation in Cryptoki requires several object creation calls. Another example of a one-to-many CDSA to Cryptoki mapping is `CSSM_GetModuleInfo`, described below.

Mapping the CSSM API to the CSSM SPI

The Core Functions and Utility Functions from the CSSM Core Services API (which initialize the CSSM, load and verify modules, handle application queries about add-in modules, and do memory management), and the Cryptographic Context Operations from the CSSM CAPI (which create, retrieve, update, and free security contexts associated with an operation) have no analogues in the CSSM SPI. However, the CSSM API calls CSSM_CSP_Create*Context, CSSM_CSP_DeleteContext, CSSM_ModuleAttach, and CSSM_ModuleDetach may cause the CSSM to generate a call to CSP_EventNotify in the SPI, to inform a CSP that an event has occurred.

However, the cryptographic calls in the CSSM API are basically equivalent to those in the CSSM SPI. Several of the CSSM API and CSSM SPI cryptographic calls are available in two forms: either a single call can perform the operation, or a set of staged calls can be used. Aside from the API's "CSSM_" versus the SPI's "CSP_" prefix, the cryptographic function names are equivalent, as are their functional descriptions.

Typically, cryptographic calls in the CSSM SPI are passed two extra parameters, when compared to the CSSM API: a handle to the add-in CSP which will be used to perform "upcalls" to the CSSM for memory management, and a *pointer* to the cryptographic context describing the cryptographic operation. (Context *handles* are required parameters for most API and SPI cryptographic calls.)

A small number of error values returned by the CSSM SPI and the CSSM API differ. In general, if an error has the "CSP_" prefix, then it can be returned by either the CSSM or the CSP, but if the error does not have "CSP_" prefix, it will only be returned by the CSSM. For example, CSSM_CSP_STAGED_OPERATION_UNSUPPORTED may be returned by the CSSM API if it finds no pointer to this function, or by a CSP if the CSSM invokes a CSP function which does not support the form of staging requested. However, CSSM_INVALID_CONTEXT_HANDLE will only be returned by the CSSM, as the CSP does not manage context handles.

The cryptographic sessions and logon functions (logging in, logging out, and changing passwords) are equivalent in the CSSM API and CSSM SPI, and expect equivalent parameters. The module management functions of the API only roughly correspond to those in the SPI; the API calls are at a higher level.

The extensibility function of the CSSM API (CSSM_PassThrough) is mapped to the extensibility function in the CSSM SPI. Like the cryptographic operations, the function name is equivalent (with a different prefix), as is the operational description. But, the SPI requires two additional parameters: a handle to the add-in CSP, and a pointer to the cryptographic context.

Mapping the CDSA Calls to Cryptoki.

The CSSM Core Functions map to the Cryptoki General Purpose Functions because both deal with initializing, closing, or obtaining information about the CAPI itself, and not the cryptomodules behind it. Cryptoki's C_GetFunctionList call is mapped to CSSM_RegisterServices from the CSSM Add-In Module Interface Function Category, but it is important to note their differences. C_GetFunctionList allows an application to obtain a list of function pointers provided by a Cryptoki library when the library is first loaded, whereas CSSM_RegisterServices allows a CSP to make an upcall to the CSSM to register its function table with the CSSM (but not to provide this function table to the application). The CDSA application then uses a module handle to indicate to the CSSM which function table it should use. CSSM_VerifyComponents has no analog in the Cryptoki API, but token verification is possible through other methods, as described in "Handling Unique APIs" on page 10.

The CSSM Utility Functions did not map to Cryptoki, because the Cryptoki interface does not incorporate memory management. (The Cryptoki designers purposely avoided allocating memory on behalf of the caller.)

The CDSA Cryptographic Context Operations are mapped to the Cryptoki Object Management Functions, because both handle parameters to cryptographic calls. As described above, the CDSA context defines all context information for the current operation (aside from data), while Cryptoki objects define parts of the security context. However, since both sets of functions provide interfaces to functionality which manage the structures which define cryptographic operations, they are mapped to each other. For example, CSSM_CSP_Create*Context maps to C_CreateObject; CSSM_GetContextAttribute maps to C_GetAttributeValue and C_GetObjectSize; and CSSM_FreeContext, CSSM_DeleteContextAttributes, and CSSM_DeleteContext map to C_DestroyObject.[9]

CDSA's cryptographic calls map to Cryptoki's cryptographic calls in a straightforward, simple manner; many calls correspond in name and functionality. The interfaces to cryptographic functionality defined by the CSSM API and Cryptoki are very similar; [9] found that 88% of cryptographic calls mapped were either "equivalent" or "roughly equivalent". Both CDSA and Cryptoki provide calls to perform cryptographic operations in either a single call or in several staged calls. Differences in the cryptographic APIs, as well as methods for implementing calls unique to one API in the other API, are described in "API Differences" on page 7.

CDSA's Module Management Functions are mapped to Cryptoki's Slot and Token Management Functions and Cryptoki's Session Management Functions, because CDSA's CSP is roughly equivalent to the combination of Cryptoki's slot and token. Although CDSA's `CSSM_ModuleInstall` and Cryptoki's `C_InitToken` both perform off-line administrative initializations needed prior to using the token in a runtime environment, they were not mapped to each other because they perform different set-up services due to the different runtime environments of CDSA and Cryptoki. `CSSM_ModuleAttach` is mapped to `C_OpenSession`, because both establish a connection between the application and the cryptographic module before a cryptographic request is made. Similarly, `CSSM_ModuleDetach` is mapped to `C_CloseSession`, because both close a session between the application and the cryptographic module.

`CSSM_ModuleInstall` is not mapped to `C_InitToken`, because the former simply registers the module in the CSSM registry, whereas the latter initializes a token by destroying all temporary objects (permanent objects like keys built into the token will not be destroyed), and disabling normal user access until their PIN is set. (However, `AddInAuthenticate` from the Add-In Module Interface Function Category of the SPI, which is invoked by the CSSM after a module is loaded and authenticates the CSSM to an add-in service module, is mapped to Cryptoki's `C_InitToken`.) Similarly, `CSSM_ModuleUninstall` is not mapped to `C_CloseSession`, because the former simply removes the module from the CSSM registry, while the latter closes a session, automatically destroys all temporary objects, and optionally ejects the token. (A Cryptoki application cannot specify whether the token will be ejected; the driver writer chooses if the token will be ejected.)

The CSSM API's `CSSM_GetModuleInfo` maps to the CSSM SPI's `CSP_GetCapabilities` and to several Cryptoki functions (`C_GetSessionInfo`, `C_GetSlotInfo`, `C_GetTokenInfo`, `C_GetMechanismInfo`, `C_GetMechanismList`, `C_GetSlotList`, and `C_GetInfo`) because `CSSM_GetModuleInfo` returns in one call information which Cryptoki returns in several separate calls. The CDSA functions `CSSM_GetHandleUsage`, `CSSM_GetGUIDUsage`, `CSSM_GetModuleGUIDFromHandle`, and `CSSM_GetSubserviceUIDFromHandle` are examples of utilities specific to the CDSA architecture, which are not present in the Cryptoki interface.

CDSA's Logon Functions are mapped to Cryptoki's logon calls from the Session Management Function Category. Both CDSA and Cryptoki have sessions and optional authenticated login sequences. However, in CDSA the CSP may set up its own model for CSP administration, and no calls are defined for this. In Cryptoki, tokens may optionally define private objects and functions which require an authenticated login, and before a normal user can login, a security officer (SO) must initialize tokens and set a user's PIN. Thus, CDSA has no analog for `C_InitPIN`, but `CSSM_CSP_ChangeLoginPassword` maps to `C_SetPIN`.

CDSA defines the `CSSM_PassThrough` extensibility function. While no extensibility functions are explicitly defined in Cryptoki, calls may be added to the Cryptoki interface. (Cryptoki does not have a complex infrastructure, and thus does not require a call to allow functionality to "pass through" the infrastructure.)

Both CDSA and Cryptoki provide "callback functions". Callback functions allow the application developer to pass a function pointer to the `CSSM_ModuleAttach` and `C_OpenSession` function, respectively. The callback function is called by the CSP or token when particular events occur. At first glance a callback function seems similar to the CSSM SPI's `CSP_EventNotify`; both notify system components that a particular event has occurred. However, a callback function passes a message from the cryptomodule to the application (up from the cryptomodule), while `CSP_EventNotify` passes a message from the CSSM to the cryptomodule (down to the cryptomodule).

4 API Differences

As discussed above, CDSA and Cryptoki provide many similar calls to cryptographic functionality. (See [9] for a complete mapping.) However, recall the two major differences between CDSA and Cryptoki. First, CDSA defines application interfaces to several different security services, whereas Cryptoki was designed as an interface to (solely) cryptographic functionality. Thus, CDSA necessarily includes more auxiliary services, because more services are needed to manage its more complex architecture. Second, CDSA's CSSM API is designed to use either software or hardware CSPs, whereas Cryptoki was originally designed to provide a direct interface to hardware cryptographic tokens (although it can handle software tokens as well). Thus, the Cryptoki interface allows the application programmer to interact more directly with hardware cryptographic modules. For example, if a Cryptoki token has a protected authentication path, the Cryptoki interface may optionally allow the user to interface this path for authentication.[4] This section further explores the differences between the APIs.

Cryptographic Calls

Generally, CDSA and Cryptoki provide similar calls to cryptographic functionality.Tables 1 and 2 list the cryptographic calls unique to the CSSM API and Cryptoki, respectively. Note, the unique cryptographic calls

Table 1: Cryptographic Functionality Unique to Cryptoki

- `C_DigestKey:` continues a multi-part digesting operation by digesting a key
- `C_**Update`, where ** equals `DigestEncrypt, DecryptDigest, SignEncrypt,` or `DecryptVerify`. These are the "Dual-Function Cryptographic Functions", which perform two cryptographic operations simultaneously by one call, to avoid unnecessary passing data back and forth to and from a token.

Table 2: Cryptographic Functionality Unique to the CSSM API

- `CSSM_ObtainPrivateKeyFromPublicKey`: given a public key, returns a reference to a private key. An alternate implementation method for Cryptoki implementations is described in Table 6: Alternate Implementations for CDSA Calls.
- `CSSM_GenerateAlgorithmParams`: generates parameters for a context.

do not interface to significant cryptographic functionality, and some may be implemented through other means. The interfaces to cryptographic functionality defined by the CSSM API and Cryptoki are very similar; the full set of mappings found that 88% of cryptographic calls mapped were either "equivalent" or "roughly equivalent"[9].

Another difference between the CSSM API and Cryptoki interfaces is the set-up required for cryptographic calls. Cryptoki always requires a one-step cryptographic call to be preceeded by `C_*Init`, because the `C_*Init` call specifies algorithm, mode, and attributes to the token. CDSA, on the other hand, does not require a one-step cryptographic call to be preceeded by an initialization call. The algorithm, mode, and attributes are held in the context, which is passed to the CSP for all (one-step or staged) cryptographic requests. (Prior to making a cryptographic call, both CDSA and Cryptoki require calls to the interface for context or object creation. Cryptoki typically requires parameters (e.g. mechanisms) to be set as well.

It is important to note that cryptographic modules are not required to implement all functionality defined by these APIs, and both interfaces are potentially extensible.

Parameters to Cryptographic Calls

Another way in which CDSA differs from Cryptoki is the method by which application programmers specify the parameters for cryptographic operations (e.g. the algorithm, mode, and keys). In CDSA, the only information passed to CSSM API cryptographic calls, aside from the cryptographic context, is the data to be operated on (e.g. plaintext or encrypted data, a signature, or a MAC). Operations on CDSA contexts (creating, modifying, retrieving, or deleting a context) are performed by making calls to the CSSM API. For example, an application programmer must create a separate context (using the `CSSM_CSP_Create*Context` call) for each type of operation (e.g. signature or symmetric encryption). To change any part of a context, the programmer needs to set values in the `CSSMContextAttribute` structure, and then call `CSSM_UpdateContextAttributes` to update individual attributes.

In contrast, parameters to Cryptoki cryptographic initialization calls include not only the appropriate object for the mechanism chosen, but also a session handle, a mechanism pointer, and material to be operated on (e.g. plaintext data, a signature, or a MAC). An application programmer using Cryptoki must set values in a mechanism. A session handle is returned from the API when `C_OpenSession` is called. But, similar to CDSA, Cryptoki requires the programmer to use the Cryptoki API to manipulate objects; the programmer creates objects using the `C_CreateObject` call, and modifies objects by calling the `C_SetAttributeValue` call. Cryptoki calls also exist for retrieving a Cryptoki object, getting attributes of a Cryptoki object, or deleting a Cryptoki object.

CDSA's more packaged context could prevent unintentional errors by less experienced programmers, but could also cause performance degradation when using the API as a result of the greater interaction with the CDSA system required. Furthermore, Cryptoki's interface allows the more experienced programmer to "mix and match" objects and mechanisms, rather than requiring the programmer to call the API to reformat these as a context when they are to be used in the same cryptographic request.

4.Although CDSA does not directly interface to hardware devices, login using a protected authentication path may be implemented as follows:
1. Call `CSSM_GetModuleInfo` to determine whether the token supports a protected authentication path.
2. If so, prompt the user to use the token's authentication device.
3. Call `CSP_Login` with a NULL password.
4. Upon receiving a NULL login password, the CSP should retrieve the password entered in step 2 from the token's authentication device. This method was adapted from the PKCS #11 specification and used by Intel's PKCS #11 CSPs. (See "Porting Between the CAPIs" on page 10.)

Auxiliary Functionality

A third way in which CDSA and Cryptoki can be compared is the auxiliary functionality offered. Tables 3 and 4 list the auxiliary functionality unique to Cryptoki and the CSSM API, respectively. Unlike the APIs to cryptographic functionality, many calls to auxiliary functionality are unique to one API, and cannot be implemented in the other API. This results from the different architectures of CDSA and Cryptoki; much of the auxiliary functionality provided by CDSA, like memory

Table 3: Auxiliary Functionality Unique to Cryptoki

- `C_GetFunctionStatus`: A legacy function which usually returns `CKR_FUNCTION_NOT_PARALLEL`
- `C_CancelFunction`: A legacy function which usually returns `CKR_FUNCTION_NOT_PARALLEL.`
- `C_CopyObject`: Given an object handle and a session handle, this function creates a new object which is a copy of a given object.
- `C_Finalize`: Applications should call when finished with Cryptoki library. As of 11/12/98, a proposal to the Open Group suggested adding the `CSSM_Terminate` call to the CSSM API. `CSSM_Terminate` would shutdown CSSM services for the calling application, cleaning up the CSSM state associated with the application, and storing persistent application state. If accepted, this call would map to Cryptoki's `C_Finalize`, because if the CSSM is running in-process (rather than running as a server to many applications), a call to `CSSM_Terminate` would clean-up and exit the CSSM itself.
- `C_InitToken,` which initializes a token by destroying objects and denying access to normal users until their PIN is set. CDSA's `AddInAuthenticate` call verifies the integrity and identity of applications when application verification is required, but it does not erase objects on the token as part of the initialization, because CDSA does not handle token administration.
- `C_InitPIN`: initializes a normal user's PIN, which may be entered through the Cryptoki library or manually on a PINpad on the token.
- `C_WaitForSlotEvent:` waits for a slot event, such as token insertion or removal, to occur. CDSA provides callback functions for this purpose, which are invoked in the case of token insertion or removal.
- `C_GetOperationState` and `C_SetOperationState`: respectively save the information necessary to restart a cryptographic operation already underway, and restore the operation.
- `C_CloseAllSessions:` closes all sessions an application has with a token, automatically destroys all objects, and optionally ejects the token.

Table 4: Auxiliary Functionality Unique to CSSM API

- `CSSM_GetInfo`, which returns version information for all CSSM instances installed or registered on the local system to the application. `C_GetInfo` maps to `CSSM_GetModuleInfo`, in the CSSM Module Management Function Category, because `C_GetInfo` returns cryptomodule-specific information.
- `CSSM_RetrieveCounter`, which returns the value of a tamperproof clock.
- `CSSM_VerifyDevice`, which causes a cryptographic module to do self-verification and integrity testing.
- `CSP_EventNotify`, allows the CSSM to notify the CSP of an important event. (Included in the CSSM SPI only.)
- `CSSM_Load`, which loads the specified CSSM instance.
- `CSSM_ModuleInstall` and `CSSM_ModuleUninstall,` which (respectively) register or delete modules from the CSSM Registry. CDSA provides a registry of the modules available to the application programmer, but Cryptoki does not.
- `CSSM_RequestCSSMExemption`, which allows an application to request exemption from a standard built-in check performed by a CSSM component.
- `CSSM_FreeInfo, CSSM_Freelist,CSSM_Free, CSSM_GetAPIMemoryFunctions,CSSM_FreeKey ,`and `CSSM_FreeModuleInfo`, the CSSM memory management functions. The designers of Cryptoki deliberately avoided adding calls which allocated memory on behalf of the user, and thus Cryptoki has no memory management calls.
- `CSSM_VerifyComponents,` which authenticates CSSM components and verifies their integrity. Although Cryptoki tokens may be authenticated using certificates and challenge response, Cryptoki does not provide integrity checking.
- `CSSM_GetHandleUsage`, and `CSSM_GetGUIDUsage,` which return a bitmask describing services provided by the module specified by a given handle or GUID (respectively).
- `CSSM_GetModuleGUIDFromHandle` and `CSSM_GetSubserviceUIDFromHandle`, which return the module GUID or subservice unique ID (respectively) of the module identified by a given handle.
- `CSSM_SetModuleInfo`, which sets information describing a module.
- `CSSM_ListModules`, which lists all currently registered service provider modules in selected categories.

and module management, is necessary to support its more complex infrastructure. (See“General Introduction: CDSA and Cryptoki” on page 1 for a discussion on the different memory management interfaces provided by CDSA and Cryptoki.)While both CDSA and Cryptoki provide optional login/logout capabilities, CDSA allows CSPs to set up their own model for CSP administration, while Cryptoki simply requires that tokens be initialized by the security officer and requires that users login with a PIN before accessing private objects stored on the token.

As noted earlier, Cryptoki optionally allows a user to interface more directly with a hardware token. or example, if Cryptoki had a protected authentication path, the `C_InitToken`, `C_Login`, `C_InitPIN`, and `C_SetPIN` calls may optionally interface with a PINPad on the token, or another protected authentication path, on a token-specific basis. (While the `C_CloseAllSessions` call optionally allows a token to be ejected after closing all sessions with it, an application cannot specify whether the token is ejected; this is determined by the token driver code.)

Handling Unique APIs

Tables 5 and 6 describe some methods for implementing functionality unique to one API, when using the other API. These methods may be useful when porting an application from one API to the other, or when building adaptation layers between the CAPIs.

5 Porting Between the CAPIs

What steps would be involved when porting an application from one CAPI to the other? The differences discussed in Section “API Differences” on page 7 must be addressed. Necessary code modifications would include converting function names and parameter lists, for those calls that map directly, and using different call sequences when needed. For example, the CSSM must be loaded and initialized, but a Cryptoki library can only be initialized. Similarly, different function call sequences would be needed when calling cryptographic functions; Cryptoki requires `C_*Init` calls to proceed any cryptographic call, but the CSSM API only requires the `CSSM_*Init` to be called for multi-staged cryptographic calls.

Other porting issues are memory management, CSP verification, and direct hardware interfaces. An application using CDSA may use the CSSM’s memory management functions or perform its own memory management, but an application using Cryptoki must perform its own memory management. When using CDSA, cryptomodule verification and built-in security checks are performed by the CSSM or via the CSSM API interface, but when using Cryptoki, these must be implemented independently of the Cryptoki API. When porting an application from Cryptoki to CDSA, the direct hardware interfaces available in Cryptoki (e.g. waiting for token removal, or login through a protected authentication path on the token) would need to be implemented in CDSA through other methods or accessed as a `CSSM_PassThrough` function.

An alternative to porting an application is to build an adaptation layer which maps calls from one API to calls in the other API, as shown by Figures 2 and 3. Figure 2 shows an adaptation layer which maps a Cryptoki

Table 5: Alternate Implementations for Cryptoki Calls

- `C_SignRecover`: Signs data in one operation, where the data can be recovered from the signature. `C_SignRecover` encrypts data with a private key, and this can be accomplished in CDSA by calling `CSSM_EncryptData` with a private key.
- Similarly, `C_VerifyRecover` verifies a signature in one operation, where the data is recovered from the signature. `C_VerifyRecover` decrypts data with a public key, and this can be accomplished in CDSA by calling `CSSM_DecryptData` with a public key.[a]
- `C_SeedRandom`: Mixes additional seed material into the token’s random number generator. An application using CDSA may provide a seed for random number generation (either by providing a seed value, or by passing a callback function which generates a seed) to `CSSM_CSP_Create-RandomGenContext`. This function builds the context passed to `CSSM_GenerateRandom`.
- `C_CloseAllSessions`: Closes all sessions between an application and token, destroying all session objects. CDSA has no function closes all sessions for one application. However, when a call to `CSSM_ModuleDetach` closes the last session an application has with a module, all transient objects associated with this application should be removed.[b]
- `C_InitPIN`: Initializes a normal user’s PIN, which may be entered through the Cryptoki library or manually on a PINpad on the token. CDSA allows this, like other security administration tasks, to be defined as a PassThrough function.

a. The method described to adapt Cryptoki’s `C_SignRecover` and `C_VerifyRecover` to the CDSA interface was implemented in Intel’s Cryptoki adaptation layer (see “Porting Between the CAPIs” on page 10).

b. Additionally, if the `CSSM_Terminate` call is added to CDSA, then this call would perform `CSSM_ModuleDetach` several times, closing all sessions with the calling application, and removing all transient objects for this application. Thus `CSSM_Terminate` would map to Cryptoki’s `C_CloseAllSessions`.

Table 6: Alternate Implementations for CDSA Calls

- `CSSM_VerifyComponents:` Verifies all CSSM components, checking for tampering. Cryptoki tokens may be authenticated if it is distributed with a built-in certificate; the application can verify the certificate and then challenge the token to sign a time-varying message with its secret key. However, Cryptoki does *NOT* perform integrity checks on tokens.
- `CSSM_QuerySize:` Returns sizes of output data blocks for selected cryptographic operations. The output size (in bytes) of Cryptoki functions which return output from a cryptographic mechanism can be obtained by calling the cryptographic function, and passing a NULL pointer to the output buffer.
- `CSSM_QueryKeySizeInBits`: Returns actual and effective size of a cryptographic key in bits. The size of Cryptoki key objects may be obtained by calling `C_GetObjectSize.`
- `CSSM_ObtainPrivateKeyFromPublicKey`: given a public key, returns a reference to a private key. An application using Cryptoki could associate a value related to the public key with the private key, and use this value to search for the private key when necessary.
- `CSSM_DigestDataClone:` Clones a given staged message digest context with its attributes and intermediate result. This call may be implemented in Cryptoki using the following steps: (1) Clone the state of a staged digest operation using `C_GetOperationState; (2)` Create a new session; (3) Set the state of the new session using `C_SetOperationState.`
- `CSSM_GenerateMac, CSSM_GenerateMac {Init, Update, Final}, CSSM_VerifyMac,` and `CSSM_VerifyMac{Init, Update, Final}`: The single and multi-staged MAC generation and verification functions. Cryptoki's signature operations (using a symmetric key) are equivalent to CDSA's MAC calls.
- `CSSM_RetrieveUniqueId:` Returns a unique identifier to uniquely identify a cryptographic device. Cryptoki's `C_GetTokenInfo` returns a pointer to a `CK_TOKEN_INFO` structure, which contains the serial number of the token. This number is analogous to CDSA's unique identifier for cryptographic devices.
- `CSSM_PassThrough`: Given an operation ID and parameters, executes any type of operation exported by a CSP. In Cryptoki, new calls may be added (at the expense of interoperability provided by CAPIs). Cryptoki token vendors can also define their own mechanisms, but for interoperability, registration with PKCS is preferable.

Note: Information in Tables 5 and 6 was received from correspondences with D. Ecklund and M. Wood at Intel.

library to the CSSM SPI; Figure 3 shows an adaptation layer built above CDSA which allows a Cryptoki-compliant application to use CDSA-compliant cryptomodules. Intel has publicly announced their development of an adaptation layer above a Cryptoki library to make it accessible via the CSSM SPI (the approach shown in Figure 2). This work was demonstrated by Intel's Matthew Wood at the PKCS#11 and PKCS#15 Workshop on October8-10, 1998.

Both approaches allow an application which conforms to one interface to use cryptographic libraries which conform to both interfaces. The approach in Figure 2 supports cryptographic token developers, allowing them to continue building Cryptoki-compliant tokens; and CSSM vendors, allowing them to use Cryptoki-compliant tokens as cryptomodules. This approach is intuitively preferable for two reasons: first, Cryptoki's direct hardware interfaces make it a lower-level interface than the CSSM API. Second, because Cryptoki interfaces solely to cryptographic functionality, and CDSA interfaces to several security services, adapting Cryptoki-compliant cryptographic modules to the CSSM SPI fits into the CDSA model. (However, attempting to extend the Cryptoki interface to provide CDSA's non-cryptographic security services is entirely outside the scope of Cryptoki. Cryptoki-compliant applications which require CDSA's trust policy, data storage, certificate, or optional libraries should be ported to CDSA.) The approach in Figure 3 benefits application developers with legacy Cryptoki-compliant code, which do not require security services beyond cryptographic services, and application developers who prefer the Cryptoki programming style.

Figure 2: CDSA with a Cryptoki adaptation layer

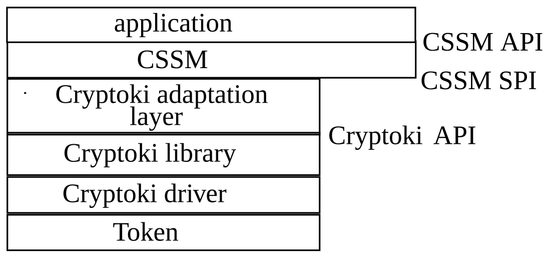
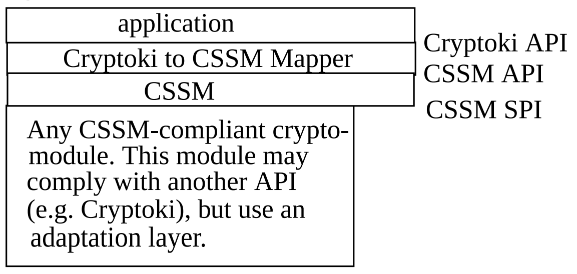


Figure 3: Cryptoki with a mapper to CDSA


application
Cryptoki API
Cryptoki to CSSM Mapper
CSSM API
CSSM
CSSM SPI
Any CSSM-compliant cryptomodule. This module may comply with another API (e.g. Cryptoki), but use an adaptation layer.

Note: Figures 2 and 3 were received from correspondences with D. Ecklund and M. Wood at Intel.

6 Conclusion

CDSA and Cryptoki are both low-level cryptographic APIs which allow application programmers to specify algorithms, modes, and attributes when requesting a cryptographic operation. Both have been recommended by the NSA Cross Organizational CAPI Team. Although CDSA is a fairly new architecture, it has been recommended by DARPA, has been accepted as a commercial standard by the Open Group, and has gained the recent support of several influential companies.

To generally compare the CSSM API calls to Cryptoki's API calls, the functionality accessed by each call in CSSM API and Cryptoki was compared, and calls were rated as equivalent, roughly equivalent, or not equivalent. (A roughly equivalent rating was given when functionality was "Implemented differently", or when due to the difference in architecture the calls had slightly different purposes. For example, CDSA's `CSSM_CSP_Create*Context` was rated roughly equivalent to Cryptoki's `C_CreateObject`.) With respect to cryptographic calls alone, 65% of cryptographic calls were equivalent, and 23% were roughly equivalent, and only 13% of the cryptographic calls had no analogue in the other API. When this rating was applied to *all* CSSM API and Cryptoki calls mapped, 33% of the calls were rated as equivalent, 25% were rated as roughly equivalent, and 41% did not have an analogue the other API. [9] It is important to note that service providers are not required to provide all functionality accessible through the API calls, and both CDSA and Cryptoki interfaces are extensible (at the expense of interoperability).

The latter set of percentages (which show the similarity between *all* of the calls mapped) reflect the fact that the CDSA has a much more comprehensive security infrastructure, and therefore provides more auxiliary services to manage this infrastructure. While Cryptoki is simply a CAPI, CDSA also defines APIs to certificate, data storage, and trust policy modules managed by the CSSM. Auxiliary services provided by CDSA (and not Cryptoki) include: CSSM self-checking, add-in module registration, user security context caching, and high level CSSM memory management. Both CDSA and Cryptoki provide some protection for sensitive objects like keys. However, when protecting security critical data or executables above the operating system, the security of the operating system must be considered when assessing the security of the overall system, as discussed in [6][7] and "Operating System Importance" on page 3.

Of the two APIs, CDSA may be preferred because of the additional service interfaces and auxiliary services it provides. For example, the trust, certificate, and data storage libraries could provide the security services needed in a networked environment with data sharing. These capabilities provide greater functionality for present work and future expansions. However, Cryptoki incorporates more capability to directly interface to hardware cryptographic tokens. The choice between the CAPIs may be further influenced by licensing, cost, current software and systems, or performance issues.

Future work could include studying sequences of calls involved in making specific cryptographic requests.

7 Acknowledgments

Denise Ecklund and Matt Wood from Intel (who led Intel's work on the PKCS #11 adaptation layer for CDSA) provided significant input on the API mappings, and on implementation methods when API calls did not directly correspond. Amy Reiss, Neal Ziring, Bill Kutz, and Steve Smalley of the National Security Agency also provided valuable input and review of this document.

8 References

[1] Open Group CDSA Specifications c707, ISBN 1-85912-194-2, 2 December 1997. http://www.opengroup.org/pubs/catalog/c707.htm

[2] Security Architecture for the AITS Reference Architecture, Draft Revision 0.62. DARPA, June 1997.

[3] Security Service API: Cryptographic API Recommendation, Updated and Abridged Edition. NSA Cross Organizational CAPI Team, NSA. October 1997. *MILCOM 97*.

[4] "New Security Standard from the Open Group Brings the Realization of High-Value E-Commerce for Everyone a Step Further". Open Group Press Release, January 6, 1998. http://www.opengroup.org/press/6jan98.htm

[5] PKCS #11: Cryptographic Token Interface Standard, An RSA Laboratories Technical Note, Version 2.01. December 22, 1997.

[6] "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments." P. Loscocco et al, National Security Agency. November 1997. *Proceedings of the 21st National Information Systems Security Conference*.

[7] "Codes, Keys and Conflicts: Issues in U.S. Crypto Policy." Report of a Special Panel of the ACM U.S. Public Policy Committee (USACM), June 1994.

[8] "Secure Computing Threats and Safeguards". Rita C. Summers. McGraw-Hill, 1997.

[9] "A Comparison of CDSA to Cryptoki". Ruth Taylor, National Security Agency. February 1999. R23 Technote #R23-TECH-001-99.