O conteúdo do presente relatório é de única responsabilidade do(s) autor(es).
The contents of this report are the sole responsibility of the author(s).

Composing Cryptographic Services: A Comparison of Six Cryptographic APIs

Alexandre M. Braga *Ricardo Dahab*
Cecília M. F. Rubira

Relatório Técnico IC–99-05

Fevereiro de 1999

Composing Cryptographic Services: A Comparison of Six Cryptographic APIs

Alexandre M. Braga   Ricardo Dahab   Cecília M. F. Rubira

State University of Campinas
Institute of Computing
P.O. Box 6176
13081-970 Campinas-SP-Brazil
phone/fax:+55+19+7885842
e-mail:{alexandre.braga,rdahab,cmrubira}@dcc.unicamp.br

Abstract

In this work we use *Tropyc*, a pattern language for cryptographic software, to evaluate a group of six Cryptographic Application Programming Interfaces (CAPIs): IBM's CCA, RSA's Cryptoki, Microsoft's CryptoAPI, Sun's JCA/JCE, X/Open's GCS-API and Intel's CSSM-API. *Tropyc* not only represents the practical issues in using the cryptographic services, but also limits the number of cryptographic mechanism combinations. We argue that these CAPIs lack the ability of cryptographic mechanism composition and, consequently, do not offer cryptographic services in an easy-to-use high-level way, resulting in low flexibility for for software reuse, an undesirable feature for cryptographic libraries targeting modern applications.

1 Introduction

Modern software systems, such as electronic commerce applications, usually have strong cryptography-based security requirements, which usually need either the composition of several cryptographic mechanisms as higher-level services or the combination of cryptographic services in a *quasi*-transparent way. Although the number of cryptographic mechanisms and their valid combinations is small, most of the presently widely used Cryptographic Application Programming Interfaces (CAPIs) do not support the full set of valid mechanism combinations. How easily an unsupported combination of mechanisms can be obtained from the supported ones is, in our opinion, an

important criteria for CAPI evaluation and is directly related not only to the CAPI's reusability but also to it's cryptographic unawareness. Cryptographic awareness is the amount of knowledge about cryptography required by the application programmer [gcs96]. We have documented the set of cryptographic mechanism combinations in a pattern language for cryptographic software, called *Tropyc* [BRD98].

In this work, *Tropyc* is used to evaluate six widely used CAPIs. Compliance to *Tropyc* means that the CAPI has the basic mechanisms required to either offer higher-level cryptographic services or instantiate the corresponding cryptographic patterns. Cryptographic service is a high-level, usually more complex, cryptographic work performed/offered by an entity, based on the corresponding cryptographic mechanisms, upon receiving requests from its clients. The cryptographic services are the following subset of the security services defined by ISO [iso98] (also called cryptographic goals [MvOV96]): data confidentiality, data integrity, authentication and non-repudiation. Accordingly, cryptographic mechanisms are the following subset of ISO's security mechanisms [iso98]: encryption, digital signatures, data integrity and authentication exchange.

We present a comparative study of six cryptographic APIs with respect to how they support *Tropyc*'s patterns. Our goal is to contrast the cryptographic mechanisms offered by widely used cryptographic APIs with the complete set of cryptographic patterns. Particularly, we want to answer three questions: (*i*) What patterns are supported by each API? (*ii*) How easily are they supported? (*iii*) Does the lack of any pattern influence the usefulness of the API in any way? Other CAPI comparisons, based on quite different criteria, can be found in [Tea97, msc].

This text is organized as follows. Section 2 summarizes the goals of modern cryptography and gives an overview of *Tropyc* and how it can be used to evaluate CAPIs. Section 3 compares the cryptographic libraries and analyzes, using function interfaces, the approaches used by CAPIs for signing data and composing cryptographic mechanisms, as well as their support to *Tropyc*. Conclusions and future work are in Section 4.

2 Cryptographic Goals and Patterns

Modern cryptography addresses four security goals [MvOV96] or services [iso98]: confidentiality, integrity, authentication, and non-repudiation. Accordingly, there are four basic cryptographic mechanisms: (*i*) encryption/decryption, (*ii*) MDC (Modification Detection Code) generation/verification, (*iii*) MAC (Message Authentication Code) generation/verification, and (*iv*) digital signing/verification. These four mechanisms can be combined in specific and limited ways to produce more high-level ones and are the building blocks for security services as well as security protocols. Confidentiality is the ability to keep information secret except from authorized users.

Data integrity is used to guarantee that information has not been modified without permission, which includes the ability to detect unauthorized manipulation. Sender (origin) authentication corresponds to the assurance, by the communicating parties, of the origin of an information transmitted through an insecure communication channel. Non-repudiation is the ability to prevent an entity from denying its actions or commitments in the future.

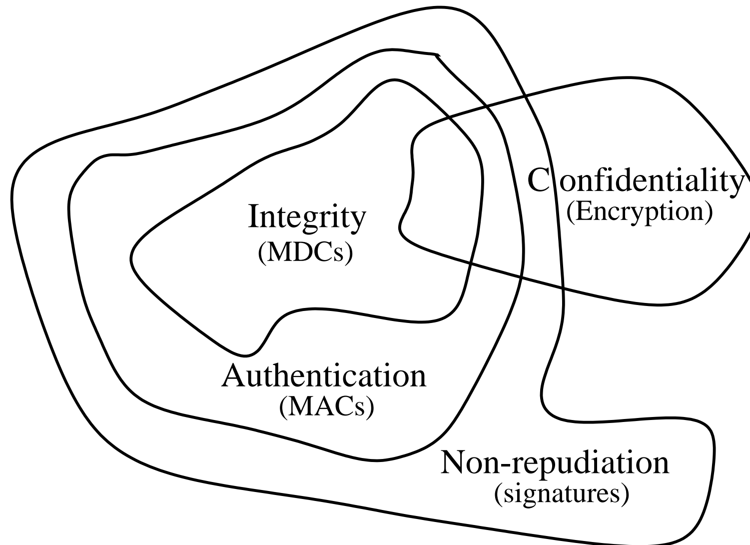

Figure 1: Relationships Among Cryptographic Services.

As shown in Figure 1, the cryptographic mechanisms corresponding to the services for data integrity, sender authentication and (digital) signatures relate to each other as follows: MACs support data integrity, signatures support both sender authentication and data integrity as well as non-repudiation. Encryption, which supports confidentiality, is orthogonal to the other cryptographic mechanisms and can be combined with each of them. It is important to notice that Figure 1 is related to cryptographic mechanisms and services, it does not necessarily work for security protocols.

2.1 Cryptographic Patterns

The basic cryptographic services are invoked in appropriate combinations with other services and mechanisms in order to satisfy application requirements. Particular cryptographic mechanisms can be used to implement the basic services. Software systems may implement particular combinations of the basic cryptographic services for direct invocation. We have proposed in [BRD98], a pattern language which addresses the proper combinations of cryptographic services, when security aspects are so important that they cannot be delegated (relegated) to the communication or storage subsystem

and should be treated by the application itself [SRC84]. The cryptographic design patterns corresponding to the basic cryptographic services and their compositions are summarized in Table 2.1. The codes in column Code of Table 2.1 are used in Figures 2 and 3 to name the cryptographic patterns.

Pattern Name	**Code**	**Scope**	**Purpose**
Information Secrecy	IS	Confidentiality	provide secrecy of information
Message Integrity	MI	Integrity	detect corruption of a message
Sender Authentication	SA	Authentication	authenticate the origin of a message
Signature	S	Non-repudiation	provide the authorship of a message
Signature with Appendix	SAp	Non-repudiation	separate message from signature
Secrecy with Integrity	SI	Confidentiality and Integrity	detect corruption of a secret
Secrecy with Sender Authentication	SSA	Confidentiality and Authentication	authenticate the origin of a secret
Secrecy with Signature	SS	Confidentiality and Non-repudiation	prove the authorship of a secret
Secrecy with Signature with Appendix	SSAp	Confidentiality and Non-repudiation	separate secret from signature

Table 2.1: The Cryptographic Design Patterns and Their Purposes.

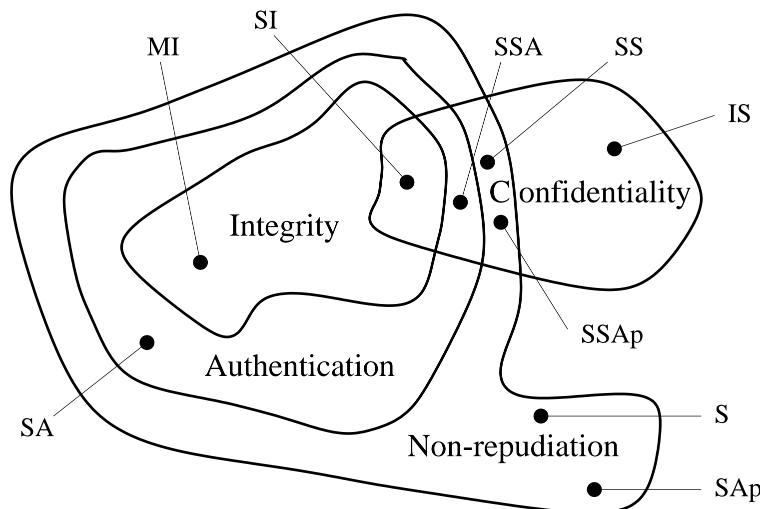

Figure 2: Pattern Distribution Over the Regions of Cryptography Services.

Figure 2 shows the distribution of cryptographic patterns over the regions of Figure 1. There is at least one pattern in each region, and two of them when an alternative, usually faster, variation exists. Because there are no uncovered regions in Figure 2, that is regions without a pattern bound to it, all the proper combinations

of the four basic cryptographic mechanisms are represented by *Tropyc*. Furthermore, there are no new combinations available. This fact is supported by the matrix of Figure 3.

Additionally, *Tropyc* documents both the use and appropriate combination of cryptographic mechanisms in order to accomplish not only the basic cryptographic services, but also the high-level composed ones. In fact, the combined patterns can be viewed as high-level services able to increase the ease of use of cryptographic libraries and APIs. CAPIs should offer not only the basic four mechanisms, but also their compositions. From a programmer point of view, CAPIs can support the composed cryptographic patterns in a variety of ways, ranging from explicit programmer-made composition of basic mechanisms to transparent composition hidden in high-level, not necessarily programmer-friendly, interfaces.

	SA	IS	S	MI	SSA	SI	SS	SAp	SSAp
SA	□	SSA	⇧	⇦	□	SSA	⇧	⇧	□
IS	SSA	□	SS	SI	□	□	□	SSAp	□
S	⇦	SS	□	SAp	SS	SSAp	□	□	□
MI	⇧	SI	SAp	□	⇧	□	⇧	⇧	□
SSA	□	□	SS	⇦	□	⇦	⇧	SSAp	□
SI	SSA	□	SSAp	□	⇧	□	SSAp	SSAp	□
SS	⇦	□	□	⇦	□	SSAp	□	SSAp	□
SAp	⇦	SSAp	□	⇦	SSAp	SSAp	SSAp	□	□
SSAp	□	□	□	□	□	□	□	□	□

□ Not significant

⇧ Supported by Column

⇦ Supported by line

Figure 3: Matrix of Cryptographic Services Combinations.

The matrix of Figure 3 shows how to obtain valid combinations of mechanisms. The full set of mechanisms in columns are combined with the same set in rows. A square-marked position means that the combination does not add any new cryptographic feature to any of its generators. A left-arrow-marked position means that the resulting combination is already implicitly supported by the row generator. An up-arrow-marked position means the combination is already implicitly supported by

the column generator. The valid combinations are marked with the codes naming the patterns. Three Observations emerge from that figure: (*i*) the number of distinct valid positions (combinations) is small; (*ii*) there are alternative ways of reaching an appropriate combination; and (*iii*) it is not necessary to add either columns or rows to the matrix because there are no new valid combinations beyond SSAp.

Since *Tropyc* completely covers not only the cryptographic services, but also their compositions, the evaluation of CAPIs based on it is complete too. Such CAPIs should not only support the basic patterns, but also the combined ones, in order to be complete. A CAPI supporting only the basic set is considered to be less adequate for modern applications than a complete one. On the other hand, the level of abstraction for providing the combinations is another aspect to be analyzed. These two aspects, support to appropriate combinations and level of abstraction, are different approaches of evaluation, in the sense that the first is quantitative and the second is qualitative.

3 Comparative Analysis

Our analysis focuses on six well known CAPIs: IBM's Common Cryptographic Architecture (CCA) [JDK+91, LMJW93], the oldest of the group; RSA's Cryptoki [Kal95]; Microsoft's CryptoAPI [Mic96]; Sun's Java Cryptographic Architecture and its Extension (JCA/JCE) [JBK98, Oak98, MDOY98], the newest of the six; X/Open's Generic Security Service API (GCS-API) [gcs96], presently deprecated; and Intel's Common Security Services Manager API (CSSM-API) [css97].

Most of these CAPIs, except JCA/JCE and CCA, were evaluated in another comparison according to quite different criteria [Tea97]. Section 3.1 provides a general evaluation of the six CAPIs according to the cryptographic services they support. The CAPIs' support to patterns is evaluated in Section 3.4. A complete description of each CAPI is beyond the scope of this text.

3.1 General Evaluation

Object-oriented CAPIs, supported by object libraries, are easier to use, and more difficult to abuse, than those ones based on function libraries. Object libraries hid potentially harmful information and offer higher level abstractions than function-based ones. Among the analyzed CAPIs, only Sun's JCA/JCE and RSA's Cryptoki present an object-oriented design, but only the first has widely available object-oriented implementations. In both cases, classes encapsulate families of semantically related functions. The other CAPIs are collections of loose-related functions (usually with large argument lists) and data types. They are usually less friendly and more prone to programming errors than the object-oriented ones.

Despite the kind of transformation performed, both object-oriented and non-

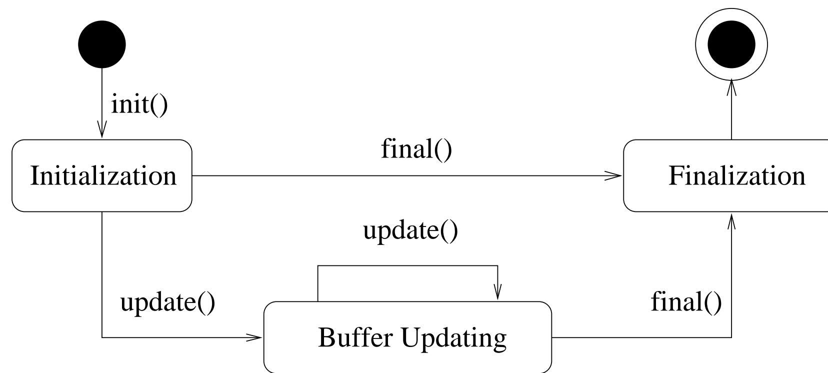

Figure 4: State Diagram for Cryptographic Transformations.

object-oriented CAPIs traverse the same state diagram, shown in Figure 4, during data transformation. There are three states: (*i*) Initialization, during which the cryptographic engine responsible for data transformation is initialized with keys and other algorithm parameters; (*ii*) Buffer Updating, in which the cryptographic engine's internal buffer is filled and some intermediary transformation is optionally performed; and (*iii*) Finalization, in which the last part of input data is accumulated and transformed. Accordingly, there are three main kinds of methods or functions responsible for state transitions: init(), update() and final().

Almost all CAPIs use byte arrays as the data structure for input and output. This requires from the CAPIs' client a great control on both data granularity and length. In fact, all CAPIs, except JCA/JCE, force their clients to worry about both block size and padding. JCA/JCE is able to work not only with byte arrays, but also with serializable objects for both signing and encryption [GS98].

The table in Figure 5 summarizes the cryptographic mechanisms the six CAPIs approach. These mechanisms can be divided in three main groups. The basic mechanisms: encryption, hashing/MDC, MAC and Signature with recovery; their compositions, formed by the Signature with appendix and the combinations of encryption with digital signatures, MDCs, or MACs; and the others, forming the group of management or auxiliary functions. Most of the CAPIs support the basic mechanisms, though a few of them support mechanism composition explicitly. Auxiliary functions are not uniformly supported. Figure 5 shows that none of the analyzed CAPIs offers routines for the complete set of mechanisms and their compositions. Furthermore, from a programmer point of view, mechanism composition is usually a difficult task, which requires a lot of knowledge about the topic. Microsoft's CryptoAPI is the most complete, but Sun's JCA/JCE is the most programmer-friendly. Section 3.2 offers a comparison of function and method interfaces for Dada signing and Section 3.3 analyses the composition of cryptographic mechanisms.

Service \ API	Cryptoki	GCS-API	JCA/JCE	CryptoAPI	CSSM-API	IBM CCA
Encryption	■	■	■	■	■	■
Encryption and Authentication				■		
Encryption and Data Integrity		■		■		
Encryption and Non-repudiation		■		■		
Digital Signature with Recovery	■					
Digital Signature with Appendix	■	■	■	■	■	■
Cryptographic Hashing/MDC	■	■	■	■	■	■
Random Numbers	■	■	■	■	■	■
Key/Key Pair Generation	■	■	■	■	■	■
Key Derivation	■	■			■	
Key Wrapping	■	■		■	■	■
Key Agreement		■	■			
Message Authentication Code				■	■	■

Figure 5: Cryptographic APIs and the Mechanisms They Support.

3.2 How CAPIs Perform Signing

This Section compares how CAPIs perform data signing and illustrates the key differences among them. These solutions go from unique powerful-but-complex functions to ease-to-use objects which encapsulate both signatures and signed data. We compare Sun's JCA, X/Open's GCS-API, RSA's Cryptoki and Microsoft's CryptoAPI. Each of them uses a different approach for data signing. Two CAPIs are not shown here: CSSM-API approaches signing similarly to Cryptoki and IBM's CCA behaves like GCS-API.

```
minor_status generate_check_value(
        session_context,                          /* output*/
        input_data,                               /* input */
        iv,                             /* input, optional*/
        chain_flag,                               /* input */
        cc,                                /* input, output */
        intermediate_result,               /* input, output */
        check_value                              /* output */
);
```

X-Open's GCS-API uses a simple function, `generate_check_value()`, for all stages of data transformation (the stages are shown in Figure 4 and are initialization, buffering and signing). Such a powerful function, whose interface is above, requires a large number of arguments, seven in total, to perform its task over arrays of bytes (`input_data`, `intermediate_result`, and `check_value`). A flag, `chain_flag`, indicates whether the transformation is in first, middle or final stage. Large arrays can be split and treated by successive calls, in such a case, the function should be explicitly fed back with `intermediate_results`. Bye the time of the final call,`check_value` contains the signature. `cc` stands for cryptographic context, which is a structure for encapsulating both sensitive data, such as keys, and the signature engine. `iv` is an optional initialization vector.

```
BOOL WINAPI CryptCreateHash(
        HCRYPTPROV hProv,                                        /* input */
        ALG_ID AlgId,                                            /* input */
        HCRYPTKEY hKey,                                          /* input */
        DWORD dwFlags,                                           /* input */
        HCRYPTHASH *phHash                                      /* output */
);

BOOL WINAPI CryptHashData(
        HCRYPTHASH hHash,                                        /* input */
        BYTE *pbData,                                            /* input */
        DWORD dwDataLen,                                         /* input */
        DWORD dwFlags                                            /* input */
);

BOOL WINAPI CryptSignHash(
        HCRYPTHASH hHash,                                        /* input */
        DWORD dwKeySpec,                                         /* input */
        LPCTSTR sDescription,                                    /* input */
        DWORD dwFlags,                                           /* input */
        BYTE *pbSignature,                                      /* output */
        DWORD pdwSigLen                                  /* input, output */
);
```

CryptoAPI's functions for signing, above, can only produce signatures with appendix and split the signing operation in two tasks: `CryptHashData()` buffers data in successive calls and generates a hash from them; `CryptSignHash()` signs the hash stored in `phHash` and returns an array of bytes containing the signature; a third function, `CryptCreateHash()`, is used for creating a hash engine, returned in `phHash`.

```
CK_RV CK_ENTRY SignInit(
        CK_SESSION_HANDLE hSession,                        /* input */
        CK_MECHANISM_PTR pMechanism,                       /* input */
        CK_OBJECT_HANDLE hkey                              /* input */
);

CK_RV CK_ENTRY Sign(
        CK_SESSION_HANDLE hSession,                        /* input */
        CK_BYTE_PTR pData,                                 /* input */
        CK_USHORT usDataLen,                               /* input */
        CK_BYTE_PTR pSignature,                           /* output */
        CK_USHORT_PTR pusSignatureLen                     /* output */
);

CK_RV CK_ENTRY SignUpdate(
        CK_SESSION_HANDLE hSession,                        /* input */
        CK_BYTE_PTR pPart,                                 /* input */
        CK_USHORT usPartLen                                /* input */
);

CK_RV CK_ENTRY SignFinal(
        CK_SESSION_HANDLE hSession,                        /* input */
        CK_BYTE_PTR pSignature,                           /* output */
        CK_USHORT_PTR pusSignatureLen                     /* output */
);
```

The four interfaces above represent the Cryptoki's family of functions for signing. `InitSign()` initializes the signing operation; after that, the function called should be `Sign()`, if data should be processed in a single part; otherwise, successive calls to `SignUpdate()` followed by a call to `SignFinal()` should take place.

These three APIs follow a procedural programming paradigm, in fact both CryptoAPI and Cryptoki use C programming language in their specifications. Such a binding to procedural languages has direct consequences over the API design. For example, arguments that could be made implicit, such as handles, in object-oriented programming, are explicit and increase the list of arguments. Two aspects are due to C-programming binds: the use of pointers to arrays of bytes and the explicit use of variables to store the length of arrays.

```
public abstract class Signature{
        public static getInstance(String algorithm);
```

```
        public static getInstance(String algorithm,String provider);
        public final void initSign(PrivateKey prvkey);
        public final void initVerify(PublicKey pubkey);
        public final void update(byte[] b);
        public final byte[] sign();
        public final boolean verify(byte[] signature);
}

public final class SignedObject implements Serializable{
        public SignedObject(Serializable o,PrivateKey pk,Signature s);
        public boolean verify(PublicKey pk,Signature s);
        public Object getContents();
}
```

The JCA's `Signature` class, above, groups a Cryptoki-like set of functions and offers the benefits of object-oriented programming. The static method `getInstance()` can be used to instantiate particular algorithm implementations; method `initSign()` initializes the engine for signing with a private key; `update()` is used for data buffering either in single or multiple calls. Method `sign()` signs buffered data and returns the signature as an array of bytes. Since object-oriented paradigm is being used, cryptographic transformations should be applied not only over bytes, but also over objects. Alternatively to the byte-based API, JCA offers the `SignedObject` class, which can be used to sign and verify (serializable) objects. JCA also offers specialized steams to handle encryption over continuous data.

These four approaches are not limited to signing. Each CAPI keeps analogous behavior for encryption, hashing and authentication, as well as for decryption and verification procedures. In summary, the four ways used by CAPIs to perform signing are the following.

1. A unique powerful function signs data in successive calls and flags determine the stage of the transformation. Feedback of intermediary results is usually required.

2. Explicit separation of signing from buffering and hashing. In this case, buffering can take multiple function class, but feedback is avoided.

3. A family of functions perform initialization, buffering and signing. Hashing is hidden and feedback is also avoid.

4. Signing is performed not only over byte arrays, but also over objects, by using a high level, usually more intuitive, programming interface.

3.3 How CAPIs Support Mechanism Composition

Among all CAPIs analyzed in this text, only GCS-API and CryptoAPI explicitly offer cryptographic mechanisms composition procedures. GCS-API still apply its approach and offer a single powerful-but-complex function to all kinds of combinations. Such a function interface, below, requires two cryptographic contexts (one for confidentiality, the other for either integrity or signature) and returns pointers to two outputs containing the encrypted data and the check value. Intermediate results still should be fed back and a flag controls whether transformation should be performed in single or multiple parts, as well as the stage of transformation.

```
minor_status protect_data(
        session_context,                                    /* output*/
        input_data,                                         /* input */
        iv,                                                 /* input,optional*/
        chain_flag,                                         /* input */
        confidentiality_cc,                                 /* input,output */
        integrity_cc,                                       /* input,output */
        intermediate_result,                                /* input,output */
        output_data                                         /* output */
        check_value                                         /* output */
);
```

Microsoft's CryptoAPI uses the function interface below to combine encryption with finger print generation mechanisms. An authentication engine, which can be either a MDC or MAC generator or even a signature engine, is passed as an argument to the function and is used internally to buffer the data before authentication. Similarly to CGS-API, a flag indicates the last or single function call. The fingerprint is obtained by calling `CryptSignHash()`, see Section 3.2, thus only data buffering is, in fact, composed.

```
BOOL WINAPI CryptEncrypt(
        HCRYPTKEY hKey,                                     /* input */
        HCRYPTHASH hHash,                                   /* input */
        BOOL Final,                                         /* input */
        DWORD dwFlags,                                      /* input */
        BYTE *pbData,                                       /* input,output */
        DWORD *pdwDataLen                                   /* input,output */
        DWORD *dwBufLen                                     /* input */
);
```

Although not allowing explicit support for cryptographic service composition, Sun's JCA reduces the complexity of this task by using SignedObjects and SealedObjects as specializations of class Serializable. When using objects for securing data, the structuring of the secure classes as specialization of serializable objects allows chaining of objects, since serializable signed objects contain other serializable objects, which can be instances of other secure objects, and so on. This is an application of the *Composite* [GHJV94] design pattern.

3.4 Pattern Support

The widespread use of design patterns has reached the design and implementation of object libraries for cryptography. For instance, Sun's JCA/JCE makes extensive use of *Factory Methods* [GHJV94] in order to allow subclasses to specify the objects to be created. There are other patterns related to security aspects of applications [YB97]. In this section, the CAPIs support to *Tropyc*'s cryptographic patterns is analyzed. Compliance to *Tropyc* means that the CAPI has the mechanisms required by an application in order to easily instantiate the patterns.

Patterns / API	Cryptoki	GCS-API	JCA/JCE	CryptoAPI	CSSM-API	IBM CCA
Information Secrecy	Explicit Use	Explicit Use	Explicit Use	Explicit Use	Explicit Use	Explicit Use
Message Integrity	Explicit Use	Explicit Use	Explicit Use	Explicit Use	Explicit Use	Explicit Use
Sender Authentication	Not Available	Not Available	Not Available	Explicit Use	Explicit Use	Explicit Use
Signature	Explicit Use	Internal Use	Internal Use	Internal Use	Internal Use	Internal Use
Signature with Appendix	Explicit Use	Explicit Use	Explicit Use	Explicit Use	Explicit Use	Explicit Use
Secrecy with Signature	Not Available	Explicit Use	Not Available	Internal Use	Not Available	Not Available
Secrecy with Integrity	Not Available	Explicit Use	Not Available	Explicit Use	Not Available	Not Available
Secrecy w/ Sender Authentication	Not Available	Not Available	Not Available	Explicit Use	Not Available	Not Available
Secrecy w/ Signature w/ Appendix	Not Available	Explicit Use	Not Available	Explicit Use	Not Available	Not Available

Explicit Use   Internal Use   Not Available

Figure 6: Cryptographic APIs and the Patterns they Support.

The table of Figure 6 confirms the results obtained from Figure 5. Figure 6 shows again that Microsoft's CryptoAPI is the most complete CAPI. Although criticized for its great complexity, GCS-API is the second most complete. In general, the CAPIs support the four simple patterns but not the composed ones. *Signature* is usually only internally supported because it is a building block for *Signature with*

Appendix, which is usually supported in a high-level way. Modern applications with cryptography-based security requirements should use either Sun's JCA/JCE or Microsoft's CryptoAPI. The first does not have objects for composed mechanisms, but is so easy to use that mechanism composition becomes a less difficult task. The second has a less friendly interface, but possesses the functions necessary to instantiate the composed cryptographic patterns with relative transparency.

4 Conclusions and Future Work

This work uses *Tropyc*, a pattern language for cryptographic software, to evaluate Cryptographic Application Programming Interfaces (CAPIs). In this text we evaluated six widely used CAPIs and argued that, in general, they do not support cryptographic mechanism composition in an easy-to-use way. Furthermore, none of them provides transparent cryptographic mechanism composition. Modern applications, such as electronic commerce systems, possess strong requirements on composing cryptographic services, in such a way that there is a practical need for cryptographic object libraries supporting services composition. Such a deficiency can be overcame by either developing (proposing) a new CAPI or extending an existing one. In such a case the JCA/JCE is the best choice for an extensible CAPI because of its modern and flexible programming interface. We are using *Tropyc* to guide the extension of a household implementation of JCA/JCE, in order to explicitly address easy composition of cryptographic services, as well as easy instantiation of cryptographic patterns.

5 Acknowledgments

This work is partially supported by CNPq; PRONEX-Finep, grant 107/97 for Ricardo Dahab and FAPESP, grant 97/11128-3 for Alexandre Melo Braga and 96/15329 for LSD-IC-UNICAMP. We would like to thank Vesna Hassler (Distributed Systems Department, Information System Institute, Technical University of Vienna) for her opportune advising.

References

[BRD98] Alexandre Melo Braga, Cecília M. F. Rubira, and Ricardo Dahab. Tropyc: A Pattern Language for Cryptographic Software. In *PLoP'98 Conference - 5th Pattern Languages of Programs*, Allerton Park, Monticello, Illinois, USA, August 1998. Washington University Technical Report #WUCS-98-25 and State University of Campinas Technical Report #IC-99-03.

[css97] Common Security Services Manager API, draft 2.0. www.opengroup.org/public/tech/security/pki/index.htm, June 1997.

[gcs96] Generic Cryptographic Service API (GCS-API) – Base Draft 8, 1996. X/Open Preliminary Specification.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Publishing Company, 1994.

[GS98] Li Gong and Roland Schemers. Signing, Sealing and Guarding Java Objects. In G. Virgna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 206–216, Berlin Heidelberg, 1998. Springer-Verlag.

[iso89] Information Processing Systems – Open Systems Interconnection – Basic Reference Model – Part 2: Security Architecture. ISO/IEC 7498-2, 1989.

[iso98] Information Technology – Vocabulary – Part 8: Security. ISO/IEC 2382-8, 1998.

[JBK98] Jonathan B. Knudsen. *Java Cryptography.* O'Reilly, 1998.

[JDK+91] Don B. Johnson, George M. Dolan, Michael J. Kelly, An V. Le, and Stephen M. Matyas. Common Cryptographic Architecture Cryptographic Application Programming Interface. *IBM Systems Journal*, 30(2):130–150, 1991.

[Kal95] B. Kaliski. Cryptoki: A Cryptographic Token Interface, Version 1.0. www.rsa.com/rsalabs/pubs/PKCS/html/pkcs-11.html, 1995.

[LMJW93] An V. Le, Stephen M. Matyas, Donald B. Johnson, and John D. Wilkins. A Public Key Extension to the Common Cryptographic Architecture. *IBM Systems Journal*, 32(3):461–485, 1993.

[MDOY98] Robert Macgregor, Dave Durbin, John Owlett, and Andrew Yeomans. *JAVA Network Security.* Prentice Hall, 1998.

[Mic] Microsoft Corporation. Using CryptoAPI. msdn.microsoft.com/library/sdkdoc/crypto/1using_80kp.htm.

[Mic96] Microsoft Corporation. Applications Programmer's Guide: Microsoft CryptoAPI. Version 2.0, 1996.

[msc] Comparison of the Open Group's GCS-API and Microsoft CryptoAPI V1.0. www.opengroup.org/public/tech/security/gcs/ms_comp.htm.

[MvOV96] Alfred J. Menezes, Paul C. van Orschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[Oak98] Scott Oaks. *Java Security.* O'Really & Associates, 1998.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Science*, 2(4):277–288, November 1984.

[Tea97] NSA Cross Organization CAPI Team. Security service API: Cryptographic API Recomendation Updated and Abridged Edition. Technical report, The National Security Agency, Ft. Meade, Meryland, July 1997.

[YB97] Joseph Yoder and Jeffrey Barcalow. Application Security. *PLoP'97 Conference, Washington University Technical Report #WUCS-97-34*, 1997.