A JCA-based Implementation Framework for Threshold Cryptography

Yih Huang, David Rine, Xunhua Wang[†]
Department of Computer Science
George Mason University
Fairfax, VA 22030, USA
{huangyih, drine, xwang4}@cs.gmu.edu

Abstract

The Java Cryptography Architecture, JCA in short, was created to allow JCA-compliant cryptography providers to be plugged into a JCA-aware application at run time. This configurable feature makes JCA widely used and assures its success. However, the public key cryptographic service interfaces defined by JCA are based on the conventional public key cryptography, which is a single-sender-single-receiver model, and does not accommodate the group-based public key cryptography well. Especially, it does not support the threshold cryptography (TC), an important type of group-based public key cryptography, which has been shown to be a useful tool to enhance system security. As a step towards the systematic application of group-based public key cryptography, this article proposes an extension to the JCA framework to integrate threshold cryptography. Under this extension, various TC providers implementing different TC primitives can be plugged into a security application at run-time. This extension also makes it easy for a existing JCA-aware application to be migrated to use threshold cryptography. An example provider of threshold RSA is implemented under this framework extension. It is our belief that such an extension would help speed up the adoption of threshold cryptography.

1 Introduction

Group-oriented cryptography has been intensively studied recently. Different from the conventional single-sender-single-receiver public key cryptography model, the entities of group-oriented cryptography are a group of users of an organization, either a hierarchic organization [1, 3, 4, 11, 17, 23] or a flat one [7, 24, 26]. Threshold cryptography [7, 24, 26] is a branch of the group-oriented cryptography where a group of users of a flat organization [5, 6, 7] can share the responsibility of a single role. The duties of the role, such as signing a contract, decrypting a document encrypted for this role or authenticating itself as a single entity to an outside party, are taken cooperatively by a subset of the group of the users through the usage of threshold cryptography primitives, such as threshold RSA and threshold DSA.

In addition to providing confidentiality and non-repudiation, threshold cryptography can be used for other security purposes as well. For example, it is used for role separation [30], to build fault tolerant applications [2, 30, 31], as an alternative to key escrow [9] and to protect a system against insider attacks [30], which is important since very often the easiest way to break into a system is to bribe an insider [21].

However, as noted in [15], there is a big gap between the state-of-the-art security research community and the state-of-the-art security practice. Threshold cryptography is no exception: despite its great potentiality to enhance system security, it is still not widely used in real life. On the other hand, with the appearance of several well-designed frameworks [28, 29] and many compatible cryptographic service providers the application of the conventional public key cryptography has been very successful. Following this successful pattern, this article investigates the ways that would accelerate the adoption of threshold cryptography. We choose the Java Cryptography Architecture (JCA) [28], an object-oriented framework [18, 19] for cryptography services by `Javasoft`, as our platform. The goal of JCA is not to provide some concrete cryptography services, instead, it aims at a framework that would allow various implementations (providers) from different vendors to be plugged seamlessly into the framework and allow a JCA-based application to switch its cryptographic providers at run-time without changing its source codes. JCA achieves this goal by adopting a 3-layer architecture: applications, the JCA framework and cryptographic providers. On the top of this 3-layer architecture are the JCA-based applications; the JCA framework sits below these applications but above

[†]Contact author. Address correspondence to *xwang4@cs.gmu.edu*

a security provider. The JCA framework provides upward to applications with a uniform security interface consisting of a set of abstract classes, called `engine` classes, which are the abstraction of many cryptography concepts. The JCA framework also defines a downward interface, called `Service Provider Interface (SPI)` to which all security providers would supply the actual cryptographic service.

JCA is pretty successful and is widely accepted. However, the interfaces defined by JCA are based on conventional public key cryptography and JCA does not support any group-oriented cryptography well. In this article we extend the JCA architecture to integrate the threshold cryptography, one of the most important types of group-based public key cryptography. Under this extension, various TC providers can be plugged into a security application at runtime. The extension also makes it easy for those JCA-based applications to easily be migrated to use threshold cryptography to enhance system security. It is our belief that this integration would speed up the adoption of threshold cryptography.

This paper is organized as follows: Section 2 first reviews the related work. Introductions to the JCA framework and threshold cryptography are given in Section 3. In Section 4 we develop an JCA extension for threshold cryptography. Section 5 presents an example service provider under such an extension. Section 6 draws some conclusions for this paper.

2 Related Work

JCA [28] and the Common Data Security Architecture (CDSA) [29] are two cryptographic frameworks for conventional public key cryptography. Neither of them supports group-oriented cryptography.

Meanwhile, there are several implementations of threshold cryptography. However, they are aiming to implement certain type of TC primitives for special purposes and neither of them proposes a generic framework for threshold cryptography applications. [31] implements, in `C` language, the DDB94 threshold RSA primitive [8]. [2] gives an implementation, in `Java`, of the GJKR96 threshold/proactive DSA primitive [14], which is a self-contained API. [30] implements both the DDB94 threshold RSA and the GJKR96 threshold DSA in `C` language.

The work of this paper differs from the above in that, instead of focusing on a special TC primitive implementation, we give a JCA-based framework for threshold cryptography which can accommodate various TC primitives and implementations.

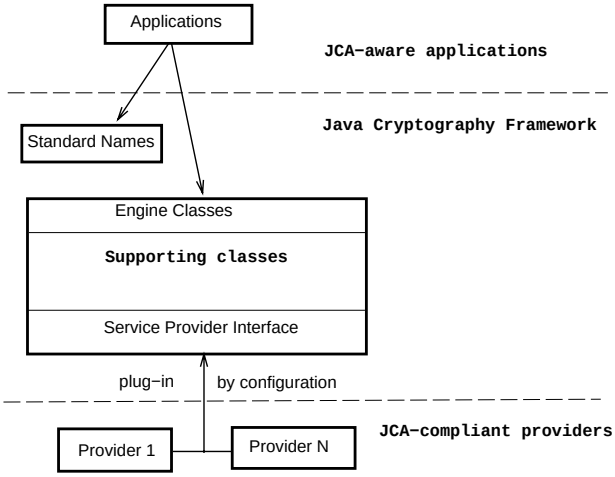

Figure 1. The JCA Architecture

3 Background

3.1 Java Cryptography Architecture

An object-oriented framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact [18]. JCA is an object-oriented frameworks to provide cryptography services. It is first introduced in the Java Development Kit (JDK) 1.1 to accommodate message digest and digital signature services, and then, is extended in JDK 1.2 to incorporate encryption, key agreement and Message Authentication Code (MAC) services, which is called the Java Cryptography Extension (JCE). Throughout this article, unless stated explicitly, we will use the JCA to denote both the original JCA and the JCE extension.

A good cryptography framework would be algorithm independent and implementation independent. To be `implementation` independent a framework should hide the details of an provider from an application and the application should not call `directly` any packages of the provider. In other words, a framework should separate an application and providers by sitting between them. In this way an application can only see an implementation-independent interface, called `upward interface`, of the framework. On the other hand, a framework should provide a downward API that can be implemented in many ways. To be `algorithm` independent the upward interface of a framework should be abstract and only correlate to generic cryptographic concepts, such as Message Digest, instead of concrete algorithms such as SHA-1 and MD5.

The JCA design follows the above paradigm and its structure is shown in Figure 1.

The JCA architecture contains three pieces: the JCA-

based applications, the JCA framework and the JCA-compliant providers. On the top are the applications built on JCA. To these applications the JCA framework provides an upward interface, called `engine` classes, along with standard names, which is a set of security algorithm names (for example, `RSA` and `MD5`). A JCA-based application only knows the engine classes and the standard names. Thus only engine class names and standard names will appear in the application's sources code and its configuration files. The JCA framework also defines a downward uniform API, called "Service Provider Interface" (SPI), to underlying cryptographic providers. This uniform interface allows a provider to be replaced at run-time.

The JCA engine classes are included in the `java.security` and `javax.security` packages. More specifically the engine classes of JCA include `MessageDigest`, `Signature`, `KeyFactory`, `KeyPairGenerator`, `KeyFactory`, `CertificateFactory`, `KeyStore`, `AlgorithmParameters`, `AlgorithmParameterGenerator`, `SecureRandom`, `Cipher`, `KeyGenerator`, `SecretKeyFactory`, `KeyAgreement`, and `Mac`. We would like to explain briefly those relevant classes here: the `Signature` is the interface for digital signature signing and verification; `Cipher` represents the interface for encryption/decryption; `KeyPairGenerator` defines the interface to generate a public/private key pair. JCA defines 2 types of keys: opaque and transparent keys. An opaque key representation is an key in which an application has no direct access to the key material that constitutes the key; a transparent representation of keys allows an application to access each key material value individually. `Key`, `PrivateKey` and `PublicKey` are defined for opaque keys while `KeySpec` for transparent keys. The `KeyFactory` engine class provides the conversions between them.

The above-mentioned JCA design allows an application to choose cryptography providers at run-time and has been proved to be very successful [27].

3.2 Threshold Cryptography

Threshold cryptography is a society-oriented cryptography [5, 6, 7]. In threshold cryptography, the message receiver/signer is not an ordinary individual, but an entity of an organization, such as a department, whose duties are *collectively* assumed by a group of users of this organization. These users share responsibilities of the entity to decrypt a message or sign a message, which is a desirable way to prevent power abuse. On the other hand, from the viewpoint of an outsider, this role is assumed by a single entity of the organization (the department in the above example). Therefore, the internal structure of the organization is kept secret from outsiders.

In threshold cryptography each entity owns a public key and the corresponding private key is shared among a group of, say n, users. Any b, $1 \leq b \leq n$, of these n users can co-decrypt or co-sign a message without reconstructing the shared private key. On the other hand, any subset with size less than t, $1 \leq t \leq n$, users can neither recover the private key nor co-sign/co-decrypt a message on behalf of the entity. So far several threshold RSA primitives [12, 8, 10, 13, 24] and threshold DSA primitives [22, 14] have been presented in the research community.

[12] presents a threshold RSA primitive with *heuristic* security and [10] gives a *provably* secure threshold RSA primitive in which each user has only one key share but the computation is done in a very complex algebraic structure. [8] applies combinatorics to develop an elegant and simple threshold RSA primitive. [13] discusses how to add *robustness* to a threshold RSA primitive and [24] integrates *proactiveness* into threshold RSA and presents a simple threshold RSA primitive. All of the above primitives could be used for both threshold decryption and threshold digital signature schemes.

[22] gives the first threshold DSS primitive. [14] presents a more efficient threshold DSS primitive. It also addresses the *robustness* issue of the threshold DSS primitive.

Threshold cryptography has been implemented to achieve fault tolerance [2, 31], role separation [30] and protection against inside attackers [30]. Potentially it can be employed by organizations such as government and companies.

However, threshold cryptography is still not widely used in the real world in spite of its advantages.

4 The Framework Extension for Threshold Cryptography

In this section, we will define a framework for threshold cryptography which will allow any compliant TC providers to be plugged into the framework and an application based on the framework extension can switch its TC providers at run-time without changing any of its source codes. It should be noted that it is not our goal to implement any specific threshold cryptography primitives.

4.1 Threshold cryptography primitive extension

A cryptographic *primitive* is a basic mathematical operation on which cryptographic *schemes* can be built [20, 25]. Conventional public key cryptography defines four types of primitives: encryption, decryption, signature and verification [25]. For example, conventional RSA has the following four primitives: RSAEP, RSADP, RSASP1 and RSAVP1 [20, 25]. A *scheme*, on the other hand, combines crypto-

graphic primitives and other techniques (such as the OAEP-encoding [25]) to achieve a particular security goal.

In threshold cryptography, for an outsider, the group of users act as a single entity, so the encryption and signature verification should be the same as those defined in conventional public key cryptography. Decryption and signature, however, will be different. Threshold cryptography introduces two new types of primitives for RSA algorithm, `TCRSADP` and `TCRSASP1`, and one new type of signature primitive, `TCDSASP`, for DSA algorithm.

We now formalize these new primitives:

- TCRSADP (B, c)

  Input 1. B: the set of users who will co-decrypt the received message
  2. c: ciphertext to be decrypted

  Output m, the corresponding plaintext or an error

  Steps the users in set B will co-decrypt the received ciphertext and recover the plaintext

- TCRSASP (B, m) and TCDSASP (B, m)

  Input 1. B: the set of users who will co-sign the message
  2. m: the message to be signed

  Output s, the signature of m by the shared group private key

  Steps the users in set B will co-sign the agreed message.

Two new engine classes, `TCCipher` and `TCSignature`, are defined to introduce these services to the JCA framework. It should be noted that these definitions should be abstract to all threshold cryptographic primitives rather than base on any concrete threshold primitives.

1. `TCCipher` defines the generic threshold decryption operation. Different from the `Cipher` engine class of the JCA framework, the TCCipher requires more information for a threshold decryption, mainly the information about the subset of the group of the users who will participate the threshold decryption.

2. `TCSignature` defines the generic threshold signature. Different from the `Signature` engine class of the JCA framework, TCSignature requires more input information from an application, mainly the information about the subset of the group of the users involved in the threshold signature.

In correspondence with the two new engine classes, two new SPI classes, `TCCipherSpi` and `TCSignatureSpi`, are defined and they should be implemented by threshold cryptography service providers.

The definition of `TCCipher` and `TCSignature` supports two type of modes: *on-line* mode and *off-line* mode. In the on-line mode, an application is required to provide the **locations** of those users (such as user 1 at the port 9897 of 129.174.1.13) necessary for one threshold computation. Such location information is passed through framework to the provider who will communicate with these users real-time and generate the final result. The final result is in turn returned to the calling application via the framework. Thus, in on-line mode, as far as the application is concerned, all computations are done in one call. The on-line mode requires that all participating users be available at some moments. On the other hand, in real life this may not be desirable sometimes. For example, some users may be willing to participate a computation but is temporarily not available (a user walks away for instance) when the computation is started. Off-line mode is designed for this case and it allows an application to provide only **who** will participate (for example, users 1, 4 and 5) for one threshold computation. In this case, the framework will only return a partial result, instead of the final one. It would be the application's responsibility to collect other partial results and combine them. Off-line mode is useful for smart-card based implementation.

4.2 The key share extension

In the conventional public key cryptography, there are two types of key, public key and private key. Public key remains the same in threshold cryptography. However, the (group) private key does not exist as a whole and is shared among a group of users. The sub keys owned by the users are called *key shares*. A *key group* is defined as a collection of all key shares of a shared (group) private key. The number of key shares by a user and the form of key shares varies from primitives to primitives.

In the extension for threshold cryptography, some new interfaces and classes are introduced for key shares and key group. `TCPrivateKeyShare` is an interface to represent a generic key share and `TCPrivateKeyGroup` is an interface to a key group. `TCKeyGroup` is defined to be the container of a public key and the corresponding `TCPrivateKeyGroup`.

4.3 Other extensions

Since the key share generation process of threshold cryptography is different from the conventional public/private key pair generation process, a `TCKeyGroupGenerator` engine class is defined and its corresponding SPI class, `TCKeyGroupGeneratorSpi`, is introduced.

The UML class diagram of the framework extension is shown in Figure 2.

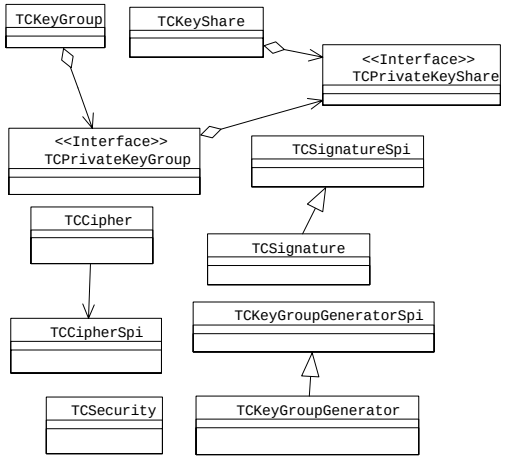

Figure 2. The UML Class Diagram of TC Framework Extension

5 An Example Provider

To test the above framework extension, we have implemented an example provider based on the DDB94 threshold RSA algorithm [8], which can be used for both threshold decryption and threshold signature.

5.1 Notations

Throughout this paper, N is used to denote the RSA modulus, which is the product of two primes, p and q. (N, e) is the group public key and d is the shared group private key. n is the size of the group and t is the minimal size of a trusted subgroup.

5.2 The DDB94 Threshold RSA key share

We will use the Abstract Syntax Notation One (ASN.1), a standard for describing data objects [16], as the expression language. Formal names in ASN.1 are written without spaces, and separate words in a name are indicated by capitalizing the first letter of each word except the first word. Furthermore, we use the Distinguished Encoding Rules (DER), a widely used encoding rule in cryptographic community, to store the key share persistently.

The DDB94 threshold RSA private key share contains the following information $\{N, e, n, t,$ key share values$\}$, which can be expressed as follows:

```
TCRSADDB94PrivateKeyShare ::=
    SEQUENCE {
        version         INTEGER,
        modulus         INTEGER, -- N
        publicExponent INTEGER, -- e
        groupSize       INTEGER,
        trustLimit      INTEGER, -- t
        keyId           INTEGER,
        subKeys    TCRSADDB94SubKeys,
}

TCRSADDB94SubKeys ::=
        SET OF TCRSADDB94SubKey

TCRSADDB94SubKey :: = SEQUENCE {
        subKeyValue  INTEGER,
        keyshareList TCRSADDB94List
}

TCRSADDB94DDB94List ::=
        SEQUENCE OF DDB94BasicList

DDB94BasicList ::= SEQUENCE {
        start INTEGER,
        end   INTEGER
}
```

5.3 The framework SPI implementation

The example DDB94 provider has the following SPI concrete classes:

- `TCRSADDB94Cipher` extends the `TCCipherSpi` and implements the DDB94 threshold RSA algorithm. It is used in the DDB94 threshold decryption and in the DDB94 threshold co-signing as well.
- `TCRSADDB94Signature` extends the `TCSignatureSpi` and provides the DDB94 threshold signature service.
- `TCRSADDB94KeyGroupGenerator` is defined to extend the `TCKeyGroupGeneratorSpi` and used to generate key shares using the DDB94 algorithm.
- `TCRSADDB94PrivateKeyGroup` extends the `TCPrivateKeyGroup` and houses a collection of `TCRSADDB94PrivateKeyShare`s defined in Section 5.2
- `TCRSADDB94KeyFactory` extends the `KeyFactorySpi` class

The UML class diagram of the DDB94 example provider is given in Figure 3.

5.4 Implementation Results

On a testing platform that comprises Microsoft NT workstations (with Pentium 233 CPU and 128M memory) interconnected by a 10 Mbps local area network, we test the

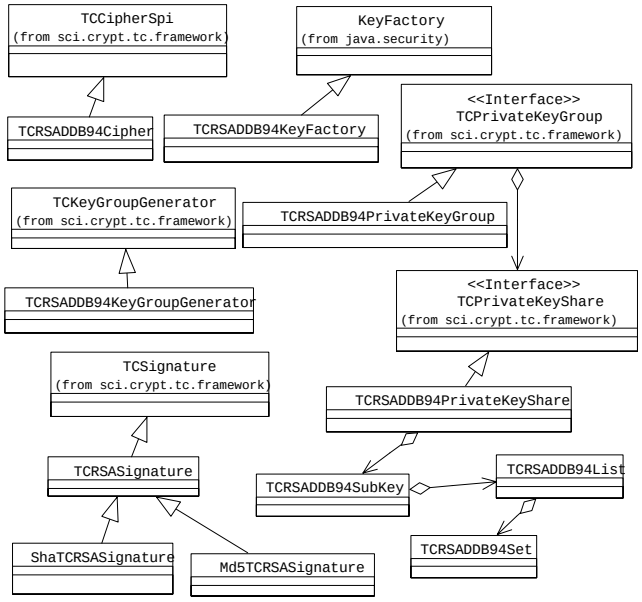

Figure 3. The UML Class Diagram of the DDB94 TC Provider

example provider under the case of (n=5, t=3). The test results are in Table 1.

Table 1. Performance of the TCRSADDB94 example provider

Key Length (bits)	Time to compute (seconds)	
	Threshold Decryption	Threshold Signature
1024	4.106	4.238
2048	10.661	10.623

6 Conclusion

Java Cryptography Architecture (JCA) is a successful object-oriented framework for conventional public key cryptography and has been widely used. In this article, we extend the JCA framework to integrate the threshold cryptography, a branch of group-oriented public key cryptography which has been shown to be a very useful tool to improve system security. Under such an framework extension, an application can easily change its threshold cryptography providers at run time without changing its source codes. Since our extension follows the JCA design principle, those JCA-based applications can easily be migrated to use threshold cryptography. An example provider is implemented to show the feasibility of the framework extension. It is our belief that this practice will help speed up the adoption of threshold cryptography.

7 Acknowledgements

The third author wish to thank Yvo Desmedt for his lectures on threshold cryptography and Hua Lin for many helpful discussions on the paper.

References

[1] G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik. A practical and provably secure coalition-resistant group signature scheme. In *Advance in Cryptology – CRYPTO 2000*, pages 255–270, Auguest 2000.

[2] B. Barak, A. Herzberg, D. Naor, and E. Shai. The proactive security toolkit and applications. In *The 6th ACM Conference on Computer and Communications Security*, pages 18–27, November 2–4 1999.

[3] M. Burmester, Y. Desmedt, H. Doi, M. Mambo, E. Okamoto, M. Tada, and Y. Yoshifuji. A structured ElGamal-Type multisignature scheme. In *Proceedings of the 3rd International Workshop on Practice and Theory in Public Key Cryptosystems (PKC2000)*, pages 466–483, January 2000.

[4] J. Camenish and M. Stadler. Efficient group signature schemes for large groups. In *Advances in Cryptology — Crypto '97*, pages 410–424, August 17–21 1997.

[5] Y. Desmedt. Society and group oriented cryptography : a new concept. In *Advances in Cryptology, Proc. of Crypto '87*, pages 120–127, August 16–20 1988.

[6] Y. Desmedt. Threshold cryptography. *European Trans. on Telecommunications*, 5(4):449–457, July-August 1994. (Invited paper).

[7] Y. Desmedt. Some recent research aspects of threshold cryptography. In E. Okamoto, G. Davida, and M. Mambo, editors, *Information Security*, volume 1396 of *Lecture Notes in Computer Science*, pages 158–173, September 1997.

[8] Y. Desmedt, G. D. Crescenzo, and M. Burmester. Multiplicative nonabelian sharing schemes and their application to threshold cryptography. In *Advances in Cryptology — Asiacrypt '94*, pages 21–32, November/December 1994.

[9] Y. Desmedt, Y. Frankel, and M. Yung. A scientific statement on the clipper chip technology and alternatives). Available at ftp://ftp.cs.uwm.edu/pub/tech_reports/desmedt-alternative-clipper.ps.

[10] Y. G. Desmedt and Y. Frankel. Homomorphic zero-knowledge threshold schemes over any finite abelian group. *SIAM Journal on Discrete Mathematics*, 7(4):667–679, November 1994.

[11] H. Doi, M. Mambo, and E. Okamoto. On the security of the rsa-based multisignature scheme for various group structures. In E. Dawson, A. Clark, and C. Boyd, editors, *Proceedings of the 5th Australasian Conference on Information*

Security and Privacy (ACISP 2000), volume 1841 of *Lecture Notes in Computer Science*, pages 352–367, July 2000.

[12] Y. Frankel and Y. Desmedt. Parallel reliable threshold multisignature. Tech. Report TR–92–04–02, Dept. of EE & CS, Univ. of Wisconsin–Milwaukee, April 1992. ftp://ftp.cs.uwm.edu/pub/tech_reports/desmedtrsathreshold_92.ps.

[13] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust and efficient sharing of RSA functions. In *Advances in Cryptology — Crypto '96*, pages 157–172, August 18–22 1996.

[14] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. In *Advances in Cryptology — Eurocrypt '96*, pages 354–371, May 12–16 1996.

[15] L. Gong and R. Sandhu. What makes security technologies relevant? *IEEE Internet Computing*, 4(6):38–41, Nov.–Dec. 2000.

[16] International Telecommunications Union – Telecommunication Standardization Sector, Recommendation X.680. *Information Technology – Abstract Syntax Notation One (ASN.1) – specification of Basic Notation*, Nov 15 1994.

[17] S. Jarecki and A. Lysyanskaya. Adaptively secure threshold cryptography: Introducing concurrency, removing erasures. In *Advance in Cryptology – EUROCRYPT 2000*, pages 221–242, May 2000.

[18] R. Johnson. Components, frameworks, patterns (extended abstract). In M. Harandi, editor, *Proceedings of the 1997 Symposium on Software Reusability*, pages 10–17, Boston, MA, 1997.

[19] R. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39 – 42, October 1997.

[20] B. S. Kaliski. Emerging standards for public-key cryptography. In I. Damgård, editor, *Lectures on Data Security*, number 1561 in Lecture Notes in Computer Science, pages 87–104. Springer, 1998.

[21] B. W. Lampson. *Computer Security in the Real World*. New Orleans, LA, USA, December 13 2000. Invited Essay to the 16th Annual Computer Security Applications Conference.

[22] S. K. Langford. Threshold DSS signatures without a trusted party. In *Advances in Cryptology — Crypto '95*, pages 397–409, August 27–31 1995.

[23] S. Mitomi and A. Miyaji. A multisignature scheme with message flexibility, order flexibility, and order verifiability. In E. Dawson, A. Clark, and C. Boyd, editors, *Proceedings of the 5th Australasian Conference on Information Security and Privacy (ACISP 2000)*, volume 1841 of *Lecture Notes in Computer Science*, pages 298–312, July 2000.

[24] T. Rabin. A simplified approach to threshold and proactive RSA. In *Advances in Cryptology, Proc. of Crypto'98*, pages 89–104, August 23-27 1998.

[25] RSA Laboratories. *PKCS #1 v2.1: RSA Cryptography Standard*, September 1999.

[26] V. Shoup. Practical threshold signatures. In *Advance in Cryptology – EUROCRYPT 2000*, pages 207–220, May 2000.

[27] Sun Microsystem. JCE cryptographic service providers. Available at http://java.sun.com/products/jce/jce12_providers.html.

[28] Sun Microsystem. Java cryptography architecture API specification & reference. Available at http://java.sun.com/j2se/sdk/1.3/docs/guide/security/CryptoSpec.html, December 6 1999.

[29] The Open Group. Common security: CDSA and CSSM, version 2. Available at http://www.opengroup.org/publications/catalog/c914.htm.

[30] X. Wang, Y. Huang, Y. Desmedt, and D. Rine. Enabling secure on-line DNS dynamic update. In *Proceedings of the 16th Annual Computer Security Applications Conference*, pages 52–58, New Orleans, Lousiana, USA, December 11-15 2000. IEEE CS Press.

[31] T. Wu, M. Malkin, and D. Boneh. Building intrusion tolerant applications. In *Proceedings of the 8th USENIX Security Symposium*, pages 79–91, 1999.