Implementation and Performance Evaluation of the RSEP Protocol on ARM and Intel Platforms

Suleyman Kondakci
Izmir University of Economics,
Faculty of Engineering & Computer Sciences
Izmir, Turkey
suleyman.kondakci@ieu.edu.tr

Gökhan Yılmaz
Izmir University of Economics,
Faculty of Engineering & Computer Sciences
Izmir, Turkey
gokhanyilmaz1988@gmail.com

ABSTRACT
We present performance evaluation of two low level implementations of the RSEP protocol. The implementations are realized both in Java and C++ languages, while the test and evaluations are performed on two different CPU architectures, ARM and Intel®. RSEP [21] is a security evaluation protocol used to assess security of remote systems over open and insecure networks such as the Internet. RSEP protocol provides an alternative approach to security test and evaluation, which mainly consists of a secure communication protocol, back-end services, and a variety of remote evaluation agents. Secure evaluation of remote entities/assets is a challenging issue with several important requirements such as interoperability, security, robustness, time-efficiency, and ease of applicability. Mobile agents running on hand-held ARM devices and Intel platforms perform the remote evaluation independent of time and location.

Categories and Subject Descriptors
C.2.0 [Data Communication]: [Security and protection];
D.4.6 [Security and Protection]: Authentication—Cryptographic controls

General Terms
Security, Languages, Performance

Keywords
Cryptography, security protocol, security evaluation

1. INTRODUCTION
Remote Security Evaluation Protocol (RSEP) was first introduced by [21] as a complete model for remote security evaluation comprised of a secure communication architecture, assets under evaluation, associated with each asset a security profile, and software services and agents that perform the evaluation over the Internet and other open networks.

The communication architecture of RSEP is based on a secure data exchange protocol utilizing a secure authentication architecture combined with the elliptic curve cryptography [20]. The remote security evaluation aims at providing seamless, location- and, time-neutral, and a proactive lifecycle risk management tasks avoiding tedious questionnaires and time-consuming on-site inspection procedures. As a unique concept, it also aims at inspiring developers and researchers to develop value-added evaluation tools, techniques and procedures that can be used by a variety of remote security assessment operations. The protocol description, requirements, methodology, communication, and evaluation procedures of the RSEP protocol can be found in [21] and [22] (evaluation agent).

First, we give a brief overview of the protocol and then present the performance evaluation of its implementations. In order to evaluate the performance of the RSEP protocol, we have implemented it to run on two different CPU architectures, ARM and Intel using two different programming languages, C++ and Java. Due to the scope of the paper, we do not analyse security aspects of the RSEP protocol, instead, the analysis of the running time efficiency of the implementation is considered here. Security analysis of RSEP should rather be considered in a separate work. The reader may refer to several other work that deal with approaches for verification, falsification, and characterization analysis of security protocols, e.g., [12, 1, 11, 5, 9, 24, 4].

In recent years, ubiquitous computing has become a very popular technology as it delivers truly useful solutions in pervasive communication environments. Especially, mobile wireless technologies based on General Packet Radio Service (GPRS), Digital RF, and Global Positioning System (GPS) are becoming today’s must-have technologies. Because of their popularity, high computational power in small sizes, and rate of the spread of these systems, we should make use of such systems in support of security evaluation and security provision to the wireless environment. Bearing this in mind, we have also implemented an additional version of the RSEP protocol to run on widely applied systems using small scale microprocessors such as ARM Cirrus EP9302 with 200MHz clock frequency. ARM-based embedded systems (http://www.embeddedarm.com) provide useful solutions for implementing ubiquitous computing devices.

1.1 Outline of the Paper
In the following, Section 2 presents an overview of related work, Section 3 deals with the overall architecture of the RSEP protocol, introduces its components, and briefly con-
siders the secure data exchange between the components. Section 4 discusses the performance of the implementations realized in C++ and Java, which run on two different platforms ARM and Intel. Finally, Section 5 concludes the paper.

2. RELATED WORK

Conventional on-site (location-based) assessment approaches provide test and evaluation functionalities in a disperse manner conducted by reformed hackers and elicited staff, but quite seldom by accredited facilities, [13] and [15]. On the contrary, combining test and evaluation tasks with proactive security operations cooperated transparently with network owners can more effectively enforce regular test and evaluations. Hence, the network owners can become more active in securing their systems become naturally more security-aware. RSEP implicitly exerts security administrators and users to execute systematical threat tracking operations in order to ensure the proactive lifecycle security.

Since RSEP performs evaluations remotely over open and insecure networks, adequate measures must be provided to securely authenticate the evaluators and nodes under evaluation, and the data must be confidentially exchanged among all principals participated in the evaluation. These measures are primarily necessary to prevent the RSEP agents from being misused as a powerful hacker tool. There exist a variety of authentication protocols, which are designed to function as cryptographic protocols, [2], and can be used in different context. As more and more systems and public services are linked through web-services, portals, and integrated web-applications, the need for a secure authentication standard that allows global authentication becomes also more apparent. Nowadays, the applied standards often issue a trusted third party to create and distribute certificates and resolve disputes. For example [16], suggests a mutual authentication protocol of two principals with a trusted server and exchange of a new symmetric key, which uses one-way functions and no encryption. In [19] a secure authentication protocol using un-cerified keys is proposed. It is claimed that the protocol is extended to counter session key compromise problems and to support repeated authentication tasks, in a more secure and flexible way, without losing its optimality. A hybrid authentication protocol for large mobile networks is proposed in [10].

Since RSEP is also designed to operate with wireless networks, this article can serve as a useful reference for comparison of RSEP’s functionality and its secure authentication [21] mechanisms with existing wireless security protocols. An authentication protocol for mobile networks is also proposed in [18], by which service providers and a key distribution center authenticate mutually. The protocol proposed [18] is based on symmetric cryptosystem, challenge-response, and hash chaining techniques.

Generally, the design criteria of inter-domain authentication protocols should include scalability, communication and computational efficiency, and the robustness of their security functionalities. Authors, in [10], discuss weaknesses of some existing protocols against the session key compromise, and then propose a new inter-domain authentication protocol. Based on public key, challenge-response and hash chaining, this approach intends to simultaneously achieve good scalability, low communication and computational cost, and resistance to session key compromise attacks. Though we did not perform analyses of eventual session key weakness of RSEP, however, it shows useful scalability, low cost, and higher computational efficiency.

Due to extremely large numbers involved in the RSA computations, there is still a need for more efficient methods in implementing of public key cryptosystems [23]. As known, the Elliptic Curve Cryptography (ECC) is based on elliptic curves defined over a finite field. The implementations presented here makes also use of ECC. There exist a number of implementations of elliptic curve cryptosystems for use in different context. ECC serves as an excellent candidate for secure embedded multimedia applications due to its small key size and high security protection [17]. However, with the performance profiling, several major bottlenecks of the ECC implementation are identified in [17]. A portable hardware architecture of the elliptic curve method of factoring, designed and optimized for applications in the relation collection of the Number Field Sieve, is described and analyzed in [14]. A generic ECC crypto-processor architecture that can be adapted to various area/performance constraints and finite field sizes is considered in [3], which also shows how to apply high level synthesis techniques to the microcontroller design.

During the last few years, a considerable effort has been devoted to the development of reconfigurable computers, machines that are based on the close interoperation of traditional microprocessors and Field Programmable Gate Arrays (FPGAs). Several prototype machines of this type have been designed, and demonstrated significant speedups compared to conventional workstations for computationally intensive problems, such as number crunching and strength analyses of cryptographic algorithms. Nevertheless, the efficient use and programming of such machines is still an unresolved problem [25]. As a hardware implementation, ECC has been often implemented using general purpose microcontrollers and FPGAs, however, a digital signal processor (DSP) can be more powerful than microcontrollers and less expensive than FPGAs [23]. [25] shows how the hardware architecture and programming model of this reconfigurable computer has influenced the choice of the algorithm partitioning strategy for this application. A detailed analysis of the control, data transfer, and reconfiguration overheads of FPGAs is also discussed by [25].

3. BRIEF OVERVIEW OF THE RSEP PROTOCOL

First, we need to give a brief description of an asset (computer, applications, information, nodes). As a common data structure, a hierarchical tree-structured information set is defined to describe and store values of security attributes of an asset. Thus, the data structure representing an asset can be used to store security attributes of an asset or attributes of a system under evaluation that contains multiple security items. A security attribute, for example, can represent either a set of security policy items or items of an operational system under evaluation. An attribute, in general, defines a specific security aspect of a given asset. For example, password strength (attribute) of a user account server is a specific aspect of the user account server, which is the asset itself, and file system of the asset (account server) is another attribute. Each asset has an attribute ID describing what the asset stands for, an asset value, w, describing
the weight of the asset, and a list of items (attributes of the asset).

As detail in [21] and [22], RSEP contains the necessary functionality to remotely monitor assets, systems, and currently supplied protection mechanisms, whether active or passive, in a timely manner. In particular, secure, timely, precise, and efficient monitoring of critical information systems and infrastructures can be achieved with this concept. In addition to the features presented in [21], the RSEP protocol can support risk management in grid networks and pervasive systems as well.

Application of the RSEP protocol enables the management of proactive lifecycle security for critical information systems. Additionally, as an important feature, the protocol enables evaluators to use hand-held devices that apply mobile wireless technologies in their remote security evaluations. The advantages of RSEP are obvious; primarily, it is based on a remote assessment approach independent of time and location, where, tests can be executed wherever and whenever needed.

3.1 Communication Architecture

The implementation presented here covers the mutual secure authentication of the communicating ends and secure data exchange between the evaluation agents and the assets under evaluation. Hence, we need to give a brief review of the elements and communication architecture of RSEP. As shown in Figure 1, the elements of the remote security assessment process are comprised of evaluation agents (EA), systems under evaluation, also called Target of Evaluation (TOE) [7], Security Profiles (SP) attached to each TOE, and an attack repository maintained by the evaluator.

![Figure 1: Architecture of the remote security evaluation](image)

Additionally, for homogeneous networks, we have an SP-service that provides back-end services by hosting a security profile or ideally a set of profiles for a given group of TOEs. An SP-service is the entry level process that exchanges security profiles with EAs and stores them securely in a disk file. We use the Common Criteria [7] terminology to describe the target of evaluation (TOE). We use the term evaluation agent (EA) to represent the evaluator, evaluation tool, or a principal. An EA is, indeed, a principal (a software agent or code) that remotely performs attacks, collects attack data, and generates evaluation information from the attack data under the control of a legitimate RSEP evaluator. That is, EAs conduct evaluations, collect and process evaluation data, and store the evaluation results into security profiles on the back-end service containers (TOEs).

To be more specific, an EA is a mobile client program that conducts the remote test and evaluation tasks utilizing the RSEP protocol. As detailed in [21] and [22], the agent can coordinate two types of assessment processes, direct and indirect (delegated), where the delegated test is performed locally on site either by the system administrator or an authorized evaluator. The results (scores) of the delegated test must be stored into the SP of the TOE under assessment, so that the EAs can access the results and finalize the test.

With the direct assessment, the EA composes the required set of attacks from the attack repository and launches the attacks in a secure way over the Internet, see Figure 1. Both the attacks and other data packets exchanged during an evaluation process are identified by the encrypted object EO, because the information is exchanged in encrypted form in order to ensure the confidentiality. Security evaluations using encrypted attack channels can present enormous opportunities for malicious and fraudulent use, which may lead to a number of questions on the liability of this type of security evaluations. However, when parties (evaluation facility and network owner) issue a legal agreement, similar to that used for penetration tests conducted by reformed hackers, some disputes can be resolved. Distribution and use of RSEP should only be allowed to authorized and accredited evaluation facilities and laboratories. In fact, while the majority of network owners would unlikely consider hiring a reformed hacker, they may find using certified and accredited evaluation facilities as a highly acceptable alternative. In short, steps of a typical security evaluation conducted by an EA are as follows:

1. Define the type of test and generate a related attack set from the attack repository,
2. securely connect to a TOE (the system under test),
3. download the SP from TOE,
4. upload and run the attack set generated,
5. save test results into a newly created SP,
6. compute the risk and update accordingly the contents of the SP,
7. upload the SP back to the TOE so that the owner of the TOE will be able to see and analyse the evaluation results.

Figure 2 shows a sample test screen of a delegated assessment conducted by an EA. During any type of assessment of a single asset (or system) many types of tests and attacks are applied, depending on the asset type and its source of threats. For example, an email server can be tested for password-strength, spam, virus-attachment, executable-code contents, and for malicious contents. That is, an asset may consist of many items (a tree of test objects) that are subject to test, e.g., a sample database server and its tested items, as shown in Figure 2.
Continuity, and Loss Control. Conceptually, tree of test objects belonging to a specific asset is illustrated in Figure 3. During the evaluation, some attacks may succeed while others fail. Each success of a repeated attack type is scored as a Bernoulli trial, resulting in either 0 or 1. The overall score $S$, varying between 0 and 5, for an asset with multiple items is computed by

$$ S = w (1 - \sqrt[n]{\frac{1}{\prod_{i=1}^{n} S_i}}). $$

where, $s_i$ denotes the score of the $i$th item of the asset, $n$ gives the number of items, and $0 < w \leq 5$ denotes the weight (importance factor) of the asset. Assuming the evaluation of a network with multiple assets, then the overall risk for the entire network is computed by

$$ R = n \left(\frac{\prod_{i=1}^{m} S_i}{\sum_{i=1}^{m} S_i}\right)^{\frac{1}{m}}, \quad (R, S_i) \in [0, \ldots, 5]. $$

In order to ensure secure authentication and secure data exchange during the evaluations, all communications are tunneled with the RSEP encryption scheme. Figure 4 shows the sequence of operations required for establishing a secure interaction between the EA and TOE. According to the RSEP specifications, each asset is associated with an SP, which also maintains current security details and required attributes of the associated asset. Prior to an assessment, the SP-service running on the TOE starts and waits for requests coming from remote EAs and delivers the required data (i.e., SPs) to corresponding EAs. Upon the termination of the evaluation, the evaluator EA will update the SP and return it to the SP-service so that the SP-service will store the returned SP in a disk file, which can be further analyzed by the system owner in order to see eventual vulnerabilities or improvements in the current security configuration. Following the evaluation, the test results are gathered and a risk level is computed by the associated EA. Before the EA returns the SP to the TOE, it stores the recently computed risk level and vulnerabilities, if any, back into the SP again. The diagram shown in Figure 5 describes the secure SP exchange and authentication operations between an EA and TOE node. Secure authentication and data exchange operations are realized by using cryptographic functions together with a secure message authentication. The public key cryptographic operations applied together with the keys $K_{UEA}$, $K_{UTE}$, $K_{REA}$, and $K_{RTOE}$, provide authentication, digital signature, and confidentiality of the data. As shown in Figure 5, each EA and TOE node generates a pair of public keys, $\{K_{REA}, K_{UEA}\}$ for the EA node, and $\{K_{RTOE}, K_{UTE}\}$ for the TOE, then they exchange their
matching public keys $K_{U_{EA}}$ and $K_{U_{TOE}}$ with each other. The secret key $K_T$, used for the encryption of the hash code HMAC, is first generated and exchanged during the active session using the Diffie–Hellman key exchange protocol.

As detailed in RFC1024 (http://tools.ietf.org/pdf/rfc2104.pdf), HMAC is used as Keyed-Hashing for Message Authentication tasks that are needed for secure authentication of messages, identities, and other types of objects. Mechanisms that provide integrity check based on a secret key are usually called "message authentication codes" (MAC). Typically, MACs are used between two parties that share a secret key in order to validate information transmitted between these parties. HMAC has been chosen as the mandatory-to-implement MAC for IP security, which is also used in other security protocols, such as SSL [26]. The message authentication is performed in two places: (1) during the exchange of security profiles (SPs), (2) during the connection establishment between evaluators (EAs) and the nodes undergoing security evaluation (TOEs). That is, SPs must be exchanged while preserving their integrity and confidentiality, and EAs must securely identify themselves to TOEs, because EAs as a trusted system run security tests on TOEs during which some confidential data of TOEs can be disclosed to unauthorized principals.

Regarding the random session key generation, we used Blum, Blum, Shub (BBS) cryptographic algorithm [6], which generates a sequence of highly random binary data defined as B bits according to the following recursive algorithm:

\[ X_0 = s^2 \mod (n), \]
\[ B_i = [(X_{i-1})^2 \mod (n)] \mod (2), \quad \text{for } i = 1, \ldots, \infty \]  

(3)

The binary data must be arranged in large number of bits/block (each block representing a single key) in order to construct a strong cryptographic key. The BBS algorithm works as follows. Given two prime numbers $p$, $q$, a factor $n = p \times q$, and a random number $s$ (seed) such that $s$ is relatively prime to $n$; meaning that neither $p$ nor $q$ is a factor of $s$. The predictability of the number generated is based on the difficulty of factoring $n$, i.e., given $n$ one needs to determine its two prime factors $p$ and $q$.

During the secure authentication, every EA presents a certificate, which is generated as a part of the SP. As detailed in [21], each SP, among others, define and store principal identities, issuer of the certificate, a legitimate evaluators ID, and a security profile of a given asset (or TOE). In other words, each asset is assigned a lightweight version of X.509–based SP containing evaluator’s unique ID, evaluator’s public key information, identity and digital signature of the issuer, time stamp, serial number, version, last evaluation date, period of validity, and digital signature algorithm (the algorithm is for signing the certificate). As known, X.509 defines an ITU-T (http://www.itu.int/) standard for public key infrastructure (PKI) and Privilege Management Infrastructure (PMI). The X.509 standard specifies, amongst others, standard formats for public key certificates, certificate revocation procedures, certification validation, and several certificate attributes, [8]. Though the X.509 digital certificate can be used as an alternative option for designing general purpose SPs and node certificates, we have designed a rather lightweight dedicated certificate for use in the RSEP protocol, which also confirms to the X.509 standards.

4. PERFORMANCE EVALUATION

Security in RSEP is provided by two cryptographic modules, secure authentication and encrypted data exchange between the evaluating node and the system under test. These modules are implemented both for ARM and Intel® platforms using two famous programming languages, Java and C++. The most complicated part of a secure communication is the authentication process, whereas the encrypted data exchange has only to do with the encryption on the sender’s side and decryption on the receiver’s side. A block diagram detailing the implementation of the secure authentication operation between an EA and TOE node is shown in Figure 5, where the message authentication is done as follows:

\[ E(K_s, (SP||HMAC(K_{U_{TOE}}, K_{R_{EA}}, SP))||E(K_{U_{TOE}}, K_s)), \]

given that HMAC is computed using the key $K_H$ and the plaintext $SP$ as

\[ HMAC(K_H, SP) = H[(K^+ \oplus \text{ipad}) || H[(K^+ \oplus \text{opad}) || SP]]. \]

Where, $K^+$ performs the padding of a sequence of zero bits to $K_H$ using the hexadecimal constants ipad and opad before the exclusive-OR operations.

The most CPU-intensive part of the entire system consists of the elliptic curve functions, the generation of the random session keys ($K_S$) used for encryption of active sessions, and the HMAC computation used for secure authentication, whereas the remaining part mostly consists of the concatenation and parsing of various data packets of the active session using relatively negligible CPU resources. As already noticed, we use two random session keys, $K_S$ for encrypting the entire session and $K_H$ for the generation of a digital signature based on the secure hash code HMAC. Though, one can use a single key (e.g., $K_H$) for both the session and HMAC with a relatively degraded security, we needed to separate authentication of the SP and session encryption operations in order to ensure secure authentication.

Figure 5: Modified RSEP encryption scheme used both for authentication and confidential data exchange.
and confidentiality. At the sender’s (EA) side, the HMAC scheme uses the EA’s private key $K_{R_{EA}}$ and TOE’s public key $K_{TOE}$ to generate the secret key $K_H$ for the encryption of the message authentication code HMAC, so that when the TOE receives the encrypted hash (HMAC) it uses its private key $K_{R_{TOE}}$ to decrypt the secret key $K_H$ before checking the integrity, and hence authenticity, of the HMAC received.

The performance analysis discussed here considers the efficiency of the implementations realized with two different programming languages running on two different machine architectures. That is, as shown in Table 1, we have implemented and tested RSEP for running both on ARM processors and Intel processors, using two different programming languages. Running times are critical for small devices (ARM-based), because they have extremely limited resources (memory, CPU) compared to other devices based on Intel processors. As summarized in Table 1, both of the implementations, C++ and Java, have shown different performance characteristics on different computer platforms. The results shown in the table are composed from the tests that have been executed on two distinct platforms; Intel Pentium T2370 with 1.73GHz CPU clock, and ARM9 Cirrus EP9302 processor with 200MHZ CPU clock.

It is worth mentioning that the Java implementation resulted in extremely slow running times both for the encryption and decryption operations. Therefore, the Java implementation realized on the ARM platform was immediately disqualified and not considered in the comparison process at all. Because, for example, the Java implementation consumed 40 seconds to encrypt a block of 256 bytes, which makes it quite difficult to consider using in real-time applications. However, the C++ implementation for ARM platforms in real-time applications.

Plot diagrams of the growth rates of the running times versus various input data sizes are shown in Figure 6. Both the ARM and Intel running times are in the logarithmic scale in order to illustrate the effect of infinite input size. Running times of both ARM an Intel are $O(n)$, i.e., linearly depending on the input data size, but, the diagram is plotted in logarithmic style, because for infinite input sizes the tendency in running times is assumed to grow as illustrated.

In fact, using ECC as the main encryption scheme has substantially improved the running time due to its ability to achieve higher protection even with a smaller key size. Although, we did not test the running times would be most probability much higher with the RSA algorithm. The pseudo-code description of the implementation is presented in Appendix A, where the readers can figure out the most CPU-intensive code parts and have a more insight into the implementation.

5. CONCLUSION AND FUTURE WORK

We have introduced the remote security evaluation protocol RSEP, and discussed a performance analysis of two low level implementations of it using C++ and Java, which are the most widely recognized programming languages. Both of the implementations were tested on two different computer architectures, a small hand-held system based on the ARM processor and an Intel Pentium-based laptop computer. Since the RSEP protocol was also aimed at being used ubiquitously on wireless systems, we were challenged to verify whether it could effectively run on small systems with limited resources. We have achieved satisfactory results of the C++ implementations both on ARM and Intel architectures, however, the Java implementation gave unsatisfactory results on the ARM-based system, while giving acceptable running times on the Intel-based systems. Furthermore, we have observed that ECC implementation in the C++ language performed well on small embedded systems due to its small key size and high security protection. Therefore, it is interesting to develop and test some new implementations that can run on other small embedded systems with limited resources such as FPGA, DSP, and emerging microcontrollers. Although, we have performed some attack tests on RSEP, an in-depth security analysis of the protocol itself should be considered as a separate work. Hence, a future work will be dedicated to the analysis of cryptographic strength and analysis of the protocol verification.

6. REFERENCES


APPENDIX

A. PSEUDO CODE REPRESENTATION OF THE IMPLEMENTATION OF FIGURE 3

Algorithm 1: Random Key Generator

Algorithm GetRandKey()
output: Random Session Key
repeat
    HugeInt \( \text{rand} = \text{BBS}() \)
    Boolean \( \text{randomness} = \text{testRandom}(\text{rand}) \)
until \( \text{randomness} = \text{true} \);
return \( \text{rand} \)

Algorithm 2: Verify Destination Certificate

Algorithm GetCertificate(String \( \text{dstID} \))
input: Destination ID
output: \( \text{X509 certificate denoting remote to ID} \)
Boolean \( \text{ver} = \text{false} \)
CERT \( \text{X509cert} = \text{Find}(\text{CERT repository, dstID}) \)
if \( \text{ver} = \text{true} \) then
    return \( \text{X509cert} \)
else
    return \( \text{false} \)
end if

Algorithm 3: ECC Private Key Generator

Algorithm GeneratePrivateKey()
output: Generated Private Key
EllipticCurveParameters \( \text{param} \)
repeat
    HugeInt \( \text{KR} = \text{GetRandKey()} \)
until \( \text{KR} < \text{param} \rightarrow n \);
return \( \text{KR} \)

Algorithm 4: ECC Public Key Generator

Algorithm GeneratePublicKey(PrivateKey \( \text{KR} \))
input: Private Key of Source
output: Generated Public Key
ECPoint \( \text{KU} \)
ECPoint \( \text{BasePoint} \)
EllipticCurveParameters \( \text{param} \)
BasePoint \( \rightarrow x = \text{param} \rightarrow Gx \)
BasePoint \( \rightarrow y = \text{param} \rightarrow Gy \)
\( \text{KU} = \text{ECMmultiply}(\text{BasePoint, KR}) \)
return \( \text{KU} \)

Algorithm 5: ECC Session Key Encryptor

Algorithm ECCencrypt(Plain \( \text{K}_S \), ECPoint \( \text{KU} \))
input: Session Key \( \text{K}_S \) and Destination Public Key
output: Encrypted session key (\( \text{K}_{SC} \))
HugeInt \( \text{K} = \text{GetRandKey()} \)
ECPoint \( \text{BasePoint} \)
EllipticCurveParameters \( \text{param} \)
\( \text{K} = \text{K mod param} \rightarrow p \)
if (\( \text{validkey(K)} \)) then
    BasePoint \( \rightarrow x = \text{param} \rightarrow Gx \)
    BasePoint \( \rightarrow y = \text{param} \rightarrow Gy \)
    ECPoint \( \text{point} = \text{ECMmultiply}(\text{BasePoint, K}) \)
    ECPoint \( \text{secret} = \text{ECMmultiply} (\text{KU, K}) \)
    Hash \( \text{h} = \text{Digest(secret)} \)
    Cipher \( \text{K}_{SC} = \text{Encrypt(K}_S, \text{h}) \)
    return Concatenate(point, \( \text{K}_{SC} \))
end if

Algorithm 6: ECC Session Key Decryptor

Algorithm ECCdecrypt(Cipher \( \text{K}_{SC} \), PrivateKey \( \text{KR} \))
input: Cipher session key (\( \text{K}_{SC} \)) of sender and private key of receiver
output: Decrypted session key (\( \text{K}_S \))
ECPoint \( \text{point} = \text{Tokenize}(\text{K}_{SC}) \)
Cipher \( \text{ciphertext} = \text{Tokenize}(\text{K}_{SC}) \)
EllipticCurveParameters \( \text{param} \)
ECPoint \( \text{secret} = \text{ECMmultiply} (\text{point, KR}) \)
Hash \( \text{h} = \text{Digest(secret)} \)
Plain \( \text{K}_S = \text{Decrypt(\text{K}_{SC}, h}) \)
return \( \text{K}_S \)

Algorithm 7: Session Key Encrypt Utility

Algorithm EncryptECC(Plain \( \text{M} \), String \( \text{dstID} \))
input: Session Key \( \text{K}_S \) and Destination ID
output: Encrypted Session Key \( \text{K}_{SC} \)
CERT \( \text{X509cert} = \text{GetCertificate(dstID)} \)
Cipher \( \text{C} = \text{ECMencrypt(M, X509cert \rightarrow KU)} \)
return \( \text{C} \)
Algorithm 8: Session Key Decrypt Utility

Algorithm DecryptECC(Cipher $K_{SC}$)

input : Encrypted Session Key $K_{SC}$
output: Session Key $K_S$

Plain $K_S = $ ECCdecrypt($K_{SC}, K_R$)
return $K_S$

Algorithm 9: Digital Signature Generation

Algorithm DS(Plain $SP$, String $dstID$)

input : Evaluator’s $SP$ and destination ID
output: Digital Signature of Evaluator

CERT $X509cert = $ GetCertificate($dstID$)
SecretKey $KH = $ ECCmultiply($X509cert \rightarrow KU, KR$)
Signature $h = $ HMAC($SP, KH$)
return $h$

Algorithm 10: Encryption of EA’s Packet

Algorithm PACK(Plain $EA$, String $ID_{TOE}$)

input : $SP$ and destination ID ($ID_{TOE}$)
output: Encrypted Packet ($Z$)

Signature $signature = $ DS($EA \rightarrow SP, ID_{TOE}$)
SessionKey $K_S = $ GetRandKey()
Plain $message = $ Concatenate($signature, EA \rightarrow SP$)
Cipher $C = $ Encrypt($message, K_S$)
$K_{SC} = $ EncryptECC($K_S, ID_{TOE}$)
Cipher $Z = $ Concatenate($C, K_{SC}$)
return $Z$

Algorithm 11: Decryption of EA’s Packet and Authentication of EA

Algorithm Authenticate(Cipher $Z$)

input : Encrypted EA Packet ($Z$) containing the EA’s ID
output: Authenticated EA by use of its $SP$

Cipher $K_{SC} = $ Tokenize($Z$)
SessionKey $K_S = $ DecryptECC($K_{SC}$)
Cipher $message = $ Tokenize($Z$)
Plain $M = $ Decrypt($message, K_S$)
Signature $receivedSignature = $ Tokenize($M$)
Plain $SP = $ Tokenize($M$)
Signature $newSignature = $ DS($SP, SP \rightarrow ID_{EA}$)
if ($receivedSignature = newSignature$) then
  | return $SP$
else
  | return $Invalid Principal$
end if