LAKE: A Server-Side Authenticated Key-Establishment with Low Computational Workload

KEMAL BICAKCI, TOBB University of Economics and Technology
BRUNO CRISPO and GABRIELE OLIGERI, University of Trento

5

Server-side authenticated key-establishment protocols are characterized by placing a heavy workload on the server. We propose LAKE: a new protocol that enables amortizing servers' workload peaks by moving most of the computational burden to the clients. We provide a formal analysis of the LAKE protocol under the Canetti-Krawczyk model and prove it to be secure. To the best of our knowledge, this is the most computationally efficient authenticated key-establishment ever proposed in the literature.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Authentication*; E.3 [**Data Encryption**]: *Public key cryptosystems*

General Terms: Security, Performance, Experimentation

ACM Reference Format:
Bicakci, K., Crispo, B., and Oligeri, G. 2013. LAKE: A server-side authenticated key-establishment with low computational workload. *ACM Trans. Internet Technol.* 13, 2, Article 5 (December 2013), 27 pages.
DOI: http://dx.doi.org/10.1145/2542214.2542216

1. INTRODUCTION

Secure Web browsing was introduced in the late 90s leveraging the HTTPS protocol [Rescorla 2000]. In the beginning, secure Web transactions were mainly enforced by banks, governments, or more generally, by all those activities that directly deal with the user's privacy and security.

In the last few years, users' privacy has been raised as a major issue for many other online services [Romanosky et al. 2011]; for example: social media, location dependent services and simple Web searches are now adopting the HTTPS protocol. Secure Web browsing guarantees the user's privacy and the server's authenticity by means of the TLS/SSL protocol [Dierks and Allen 1999]. In particular, the TLS/SSL protocol is characterized mainly by two phases: the establishment of a secret master key between the client and the authenticated server, and the subsequent encryption of all the transfered data. While confidentiality can be easily guaranteed leveraging both the newly-established master key and a symmetric encryption algorithm such as AES [Schaad and Housley 2002], the initial key-establishment is still a challenge for server-side performance [Apostolopoulos et al. 2000; Kant et al. 2000].

A server-side authenticated key-establishment is a set of procedures that eventually allow the client and the server to share a secret key and the server to be authenticated at the client. The challenge of such a protocol is at the server side [Zhao et al. 2005]: in particular, the server is requested to perform a heavy cryptographic operation for

G. Oligeri has been partially supported by the project, European CYber Security Protection Alliance (CYSPA), under the FP7-ICT European Programme.
Author's address: G. Oligeri; email: gabriele.oligeri@gmail.com.

DOI: http://dx.doi.org/10.1145/2542214.2542216

each new client, and with the increase of the number of the clients, performing such key-establishment may become computationally intensive.

Classical solutions involve specialized hardware, e.g. SSL accelerator [Chou 2002]. SSL accelerator cards can be installed into regular commodity server hardware and reduce the CPU-intensive parts of the SSL transaction. Such a solution turns out to be effective in cutting down the server workload, but this specialized hardware is still expensive. Nevertheless, SSL accelerators are currently the most effective solution [Thiruneelakandan and Thirumurugan 2011] to the increasing requests for Web security from Internet users. To guarantee higher degrees of data security, longer key lengths have been introduced, which in turn, means more load on TLS/SSL servers. Eventually, the performance of the TLS/SSL handshake directly affects the number of clients that can be served simultaneously.

Contribution. This article introduces LAKE, a new server-side authenticated key-establishment protocol that enables the server to establish a new authenticated and secure communication channel by computing only one asymmetric encryption. To the best of our knowledge, LAKE is the most efficient solution in terms of server workload. We provide a detailed formal analysis of LAKE under the Canetti-Krawczyk model and proved it to be secure. We show the results of a real client-server implementation, and we compare the performance of our solution with previously published results. Finally, we show how to integrate the proposed protocol with a puzzle-based algorithm in order to make it robust to the denial of service attack.

2. RELATED WORK

Rebalancing the TLS/SSL server workload has been studied in two previous works: Bicakci et al. [2006] and Castelluccia et al. [2006]. The latter proposed the so called Client-Aided RSA (CA-RSA), which takes advantage of the Chinese Remainder Theorem in order to shift some computational burden from the server to the client. The authors provided measurement results from an experimental setup based on OpenSSL library: they determined the upper bound on the number of SSL requests by measuring the number of RSA decryptions a server can perform within a given time frame. Their results showed CA-RSA speedups on the server side of 11.33 and 19.12 times for RSA-1024 and RSA-2048, respectively.

Although their work definitely improves the server performance, Bicakci et al. [2006] prove that it is possible to accomplish a server-side authenticated key-establishment even more efficiently on the server side. Such work leverages Online/Offline signatures in order to shift the greatest part of the computational workload offline, i.e., when the server is idle, or even to other machines. On the other side, the online computational workload adds up to only one RSA public key encryption and therefore is fast and efficient. Their preliminary results come from the OpenSSL speed-test suite and suggest that the server side can achieve speedups of 16 and 33 times using RSA-1024 and RSA-2048, respectively.

One-time signatures, introduced even before RSA signatures, allow signing only one message per one public key [Lamport 1979; Rabin 1978]. The scheme has been subsequently improved, allowing multiple signatures per one public key [Merkle 1987]. Lamport developed a framework to obtain a one-time digital signature from a one-way hash function [Lamport 1979]. The previous approach has been revisited by Even et al. [1989]: one-way functions are obtained from the DES encryption algorithm [Des 1977] and used in order to map blocks of the private key into corresponding blocks of the public key, and finally, the message is signed by revealing the corresponding blocks of the private key. In its basic construction, the previous approach involves an online

computation overhead of one only DES encryption and a transmission overhead of a 3k length signature for a message of k bits.

Shamir and Tauman [2001] subsequently proposed an improved Online/Offline signature scheme. The authors proposed the hash-sign-switch signature based on chameleon signatures, which in turn leverages trapdoor hash functions. Chameleon signatures are signatures that commit the signer to the contents of the signed message but do not allow the recipient of the signature to convince third parties that a particular message was signed, since the recipient can change the signed message to any other message of his choice. Authors leveraged such behavior in order to sign offline a temporary message and resign the actual one, leveraging the trapdoor of the hash function. The online computational overhead sums up to 0.1 modular multiplication while the transmission overhead is constituted by a random number and a digital signature.

Another interesting solution to online/offline signature comes from Guo and Mu [2008]. Authors proposed an optimal message signing procedure, namely O-3 signature, that does not need online computations. On the other side, the baseline approach needs the transmission of both n signatures and a random number. Moreover, the authors propose to leverage *batch verification signature*, i.e., where the cost for n signatures of different messages is less than conducting them one-by-one. *Signature aggregation*, i.e., where n different signatures on different messages can be aggregated into a single signature. They observe that a short provably-secure signature scheme satisfying the aforementioned properties can be found in Boneh et al. [2001]. Adopting such a scheme allows one to sign a message without online computations and only 320 bits as communication overhead. Nevertheless, batch verification signature, introduced by Fiat [1997] assumes nonstandard RSA procedures, and therefore, it is not compliant with current PKI infrastructures.

Recently, Yao and Zhao [2013], Ming and Wang [2010], and Liu et al. [2010] have shown that online/offline signatures can be applied to resource constrained devices, such as smart cards and wireless sensor networks, to effectively implement secure signature schemes.

3. PROBLEM STATEMENT

We consider a scenario constituted by a set of clients that want to establish a secure and authenticated access to a remote service. Figure 1 shows the reference scenario with the involved entities. Each client $\{C_1, \ldots, C_5\}$ wants to access one of the application servers in the cloud. This is usually achieved in two steps, during the first step the *authentication server* (Auth Server, hereafter Server) recognizes and grants the client the rights to access the *application servers* in the cloud (App Server), while during the second phase, the client directly accesses the application servers. In particular, each client-server pair establishes an *authenticated secret shared key*, hereafter K_s, to be used to secure the subsequent communications. Authentication is achieved by using a public key infrastructure (PKI). The client can always verify the public key authenticity of the server by leveraging a trusted third party, i.e., a certification authority, hereafter CA. In the following, we assume that the server public key, hereafter PK, is *signed* by CA ($\overline{PK}$) and the client can always verify PK authenticity leveraging both CA and $\overline{PK}$.

Generally speaking, the most accepted solution to this problem is constituted by the TLS/SSL handshake. Unfortunately, TLS/SSL is not efficient from the server-side point of view. This is a serious problem when the server is resource constrained [Potlapally et al. 2006; Shin et al. 2009], or more generally has to deal with many clients [Coarfa et al. 2006]. In fact, as it will be clear in the following, TLS/SSL

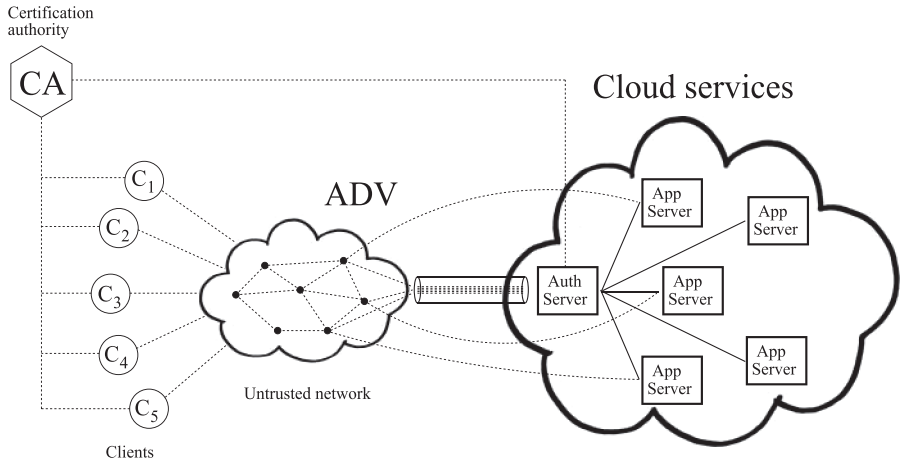

Fig. 1. Reference scenario. Clients need to be granted by the authentication server (Auth Server) in order to subsequently access the application servers (App Server).

handshake involves computationally heavy cryptographic procedures (at the server side) that may be difficult to deal with when the number of clients is high [Qing and Yaping 2009; Shacham and Boneh 2001]. Moreover, we observe that other scenarios are affected by the performance of the TLS/SSL authentication: e.g., Shen et al. [2012] studied the impact of TLS on SIP server performance and showed that using TLS can reduce the performance by up to a factor of 17 compared to the typical case of SIP-over-UDP.

Adversary model. We envisage a powerful adversary $\mathcal{ADV}$ able to control any of the node in the network of Figure 1. $\mathcal{ADV}$ can inject new messages into the network, mimic one of the existing clients, and finally eavesdrop all the exchanged messages. The adversary cannot gain control of the server, but can compromise one or more clients in order to mount a denial of service attack on the server, i.e., running multiple LAKE protocol instances on the same server.

Although our solution can be easily extended in order to authenticate both the client and the server, in this work we focus on authenticating only the server side.

3.1. Definitions

Table I shows a resume of the symbols and acronyms used throughout this article.

3.2. TLS/SSL Handshake

TLS/SSL is the most widely used protocol to ensure secure communication between a client and a server; more precisely, in this work we are interested in the initial RSA-based handshake phase, which has the goal of establishing a secret master key to be used during the TLS/SSL session.

The TLS/SSL initial key-establishment is based on RSA procedures, i.e., it leverages RSA asymmetric cryptographic primitives in order to securely converge to a shared secret key between the client and the server.

Let (SK, PK) be a pair of private and public keys, respectively, in the RSA model, obtained by the server running the algorithm G, i.e., $(SK, PK) \leftarrow G$. In the following, we assume that the client can verify the authenticity of PK relying on a trusted third party (CA belonging to a PKI).

Table I. Notation Summary

CA	Certification authority
PK	Server RSA Public/Verification Key
SK	Server RSA Private/Signing Key
pK	Client RSA Public Key
sK	Client RSA Private Key
K_s	Secret master key
pk	OTS verification keys
sk	OTS signing keys
G, S, V	RSA-based signature scheme
g, s, v	OTS-based signature scheme
$H(\circ)$	SHA-1 hash function
$\{\circ\}_{PK}$	RSA-based encryption
$\{\circ\}_{SK}$	RSA-based decryption
$S_{SK}(\circ)$	RSA-based signing procedure
$\overline{PK}$	Digital signature of the server public key PK
$V(m, \Sigma)$	RSA-based verification procedure for the message m with signature Σ
l	Symmetric key length
L	RSA Private key length

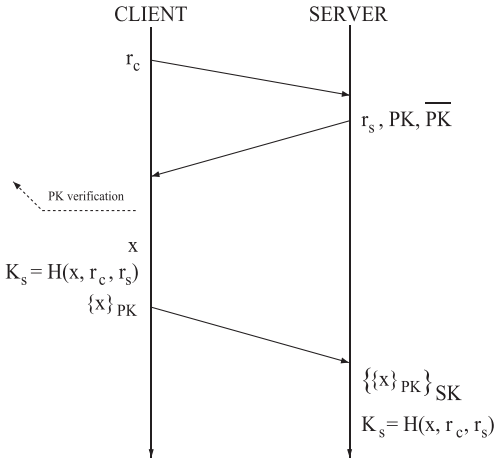

Fig. 2. A sketch of the TLS/SSL handshake.

Let $\{x, r_c, r_s\} \stackrel{\$}{\leftarrow} \mathbb{Z}$ be random integers: we use notation $x \stackrel{\$}{\leftarrow} \mathbb{Z}$ in order to generate a pseudo-random value from $\mathbb{Z}$ and assign it to x. Let also $\{\circ\}_{PK}$ and $\{\circ\}_{SK}$ be the RSA-based encryption/decryption procedures, respectively, such that $\{\{x\}_{PK}\}_{SK} = \{\{x\}_{SK}\}_{PK} = x$. A sketch of the TLS/SSL key-establishment protocol is depicted in Figure 2. The handshake protocol is initiated by the client, which generates r_c and sends it to the server (client hello packet). Subsequently, the server does the same: generates r_s and sends it back to the client together with its own certificate (server hello packet), i.e., the public key PK and a digital signature of it ($\overline{PK}$) signed by the trusted third party CA. In this way the client can leverage $\overline{PK}$ in order to verify the authenticity of PK with CA. Subsequently, the client generates the premaster key x and computes the final secret key K_s hashing all together the previous random values, i.e., $K_s = H(x, r_c, r_s)$, where $H(\circ)$ is a one-way hash function such as SHA-1 [Eastlake and

Table II. RSA-Based Computation Performance

RSA Decryption	Slow
RSA Encryption	Fast
RSA Signing	Slow
RSA Verifying	Fast

Jones 2001]. Now, the client sends the premaster key x encrypted with the server public key PK to the server. Finally, the server retrieves x by means of SK, and computes the shared secret key K_s, hashing all the previous random values x, r_c, and r_s, respectively.

We highlight how such a key-establishment protocol guarantees server-side authentication because the server provides the client with its own certificate, i.e., $< PK, \overline{PK} >$, and at the other side, the client verifies the authenticity of the received PK (from the purported server) with a trusted CA.

3.3. TLS/SSL Handshake Workload

In order to estimate the workload of the TLS/SSL key-establishment, we analyze the computational load of the most important cryptographic steps in the protocol of Figure 2. Starting from the server side, we observe a hash computation and an RSA private key decryption; while at the client side, we observe a hash computation and a public key encryption. Moreover, the client has to perform a signature verification in order to trust the public key PK received from the server.

Although we will discuss the performance issues in detail later, Table II provides a simple comparison among the most common cryptographic operations provided by the RSA algorithms. Generally speaking, we consider as fast, the operations that involve short exponents, such as the public key, while we consider as slow, the operations that involve large exponents, such as the private key. Therefore, RSA decryption and signing, which involve private key exponentiations are slow, while RSA encryption and verification, which involve public key exponentiations are fast.

Recalling Figure 2, we observe how the client workload is negligible, in fact it sums up to a hash computation, an RSA signature verification, and finally, a public key encryption. On the other side, the server workload is high; in particular, the server has to perform an RSA private key decryption for each TLS/SSL key-establishment.

4. OUR SOLUTION IN BRIEF

In this work we propose LAKE, a new server side authenticated key-establishment protocol that does not require any online heavy crypto operations. In our solution, the server performs only one online RSA encryption, while all the heavy operations can be precomputed offline. The key point of our solution relies on reversing the standard TLS/SSL handshaking, allowing the server to generate the secret key K_s, and sending it to the client encrypted with the client public key, hereafter pK. Nevertheless, such an approach has two main drawbacks: (1) the server is not authenticated and (2) the online RSA encryption performed by the server can be leveraged by malicious clients that want to mount a denial of service (DoS) attack on the server.

As it will be clear in the following, in order to authenticate the secret key K_s encrypted with the client public key pK, i.e., $\{K_s\}_{pK}$, the server could afford a classical RSA-based signature computation, but such a solution is not acceptable, because in our scenario, we want to avoid any heavy RSA computations at server side, i.e. recall Table II. We propose to adopt *online/offline signatures*, hereafter on/off signatures; in this way, heavy computations can be performed offline or even by other machines,

while the online workload sums up to only one RSA encryption. Moreover, we avoid DOS attacks at the server side by asking the client to solve a puzzle before starting the key-establishment procedure.

Finally, we summarize the two main ingredients we need in order to perform a server side authenticated key-establishment that does not rely on slow cryptographic operations (at server side) and it is robust to DoS attacks.

— *On/Off signatures*. They allow signing a message experiencing a low online computational overhead by leveraging the precomputed offline signatures. In our model, we assume offline signatures are securely stored and retrieved when new clients need to run the key-establishment protocol.
— *Puzzles*. Puzzles are well-known techniques to prevent a peer to perform a DoS attack. In our case, we want to avoid malicious clients from performing DoS attacks on the server by asking many public key encryptions. In order to avoid this, the server asks the client to solve a computationally expensive operation (puzzle), and therefore only motivated clients that really want to establish a secure connection go through this procedure.

TLS/SSL backward compatibility and practical deployment. First, we observe that LAKE needs no more than the standard cryptographic functions also used by TLS/SSL, and therefore, in the following we assume the availability of the standard *openssl* library.

In order to be easily deployed and coexist with the existing security infrastructures LAKE provides TLS/SSL backward compatibility. For this, LAKE needs a few changes in the client's hello packet. In LAKE, the client starts the protocol as in the standard TLS/SSL handshake. Clients who wish to negotiate with earlier protocols send a client a hello message using the standard TLS/SSL hello format. Then, server will respond with an SSL or TLS server hello. Clients who wish to communicate to servers with LAKE should send a client hello message having an appropriate version field to note that they support it. If the server supports only older versions, it will respond with a server hello message accordingly; if it supports LAKE, with a LAKE server hello. The negotiation then proceeds as appropriate for the negotiated protocol.

LAKE assumes the clients deal with public/private key pairs. Nevertheless, we observe that in LAKE (without client authentication), the client private keys do not need to be long term certified keys. They can be periodically erased and regenerated. As a result, managing keys in LAKE is fundamentally different than managing keys in SSL protocol with client authentication. On the extreme, the clients could generate keys per each SSL session (which brings additional performance penalty, though).

Finally, we observe that TLS/SSL protocol is used not only for Web traffic, but as a security protocol for many other possibly custom built client/server applications. Since in these new applications, the client software needs to be sent and installed by the clients anyway, we envisage delivering LAKE as part of the client application.

5. ONLINE/OFFLINE SIGNATURES

In this section we recall a general framework for implementing an Online/Offline signature scheme [Even et al. 1989].

Let $m : \{m_1, \ldots, m_l\}$ be a message of l bits, with $m_i \in \{0, 1\}$ and $i \in [1, l]$.

Let (G, S, V) represent an ordinary signature scheme [Rivest et al. 1978], and let (g, s, v) represent a one-time signature scheme [Krawczyk and Rabin 2000; Lamport 1979], where (G, g), (S, s), and (V, v) are the generation, signing and verification algorithms, respectively.

Let (SK, PK) be a pair of signing and verification keys respectively, obtained running the G algorithm $(SK, PK) \leftarrow G$. For the sake of simplicity, we use the same notation

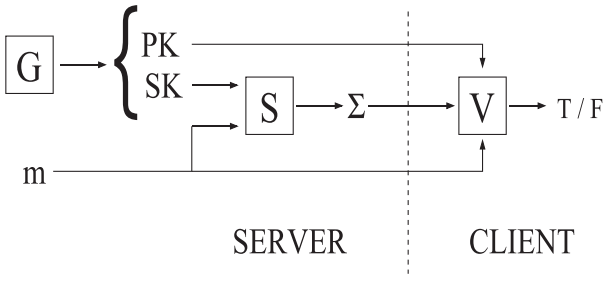

Fig. 3. Signing and verification of a message with an ordinary signature scheme.

for both the server public key and the verification key, PK. Figure 3 shows the details of an ordinary client-server message verification procedure. The client assumes the message m as trusted if and only if the verification algorithm V receives as input the signature Σ and the verification key PK, $V_{PK}(m, \Sigma)$; where the signature Σ must be generated by the signing algorithm S with input m and signing key SK, $S_{SK}(m) = \Sigma$.

Let (sk, pk) be a pair of signing and verification keys respectively, obtained running the g algorithm, $(sk, pk) \leftarrow g$. Let $v_{pk}(m, \sigma)$ be successful if and only if $\sigma = s_{sk}(m)$.

Online/Offline signature schemes are characterized by two phases: an *offline computation*, in which the server computes a trusted (message-independent) signature, Σ, yielding

$$\Sigma = S_{SK}(pk) \tag{1}$$

and an *online computation*, in which the server computes a fast (message-dependent) signature, σ, yielding

$$\sigma = s_{sk}(m). \tag{2}$$

Generally speaking, the off-line phase (Eq. (1)) is constituted by a standard (slow) signature scheme, RSA based, and outputs a trusted signature[1] Σ, i.e., pk is signed with the signing key SK. On the other side, during the on-line phase (Eq. (2)), the server signs the message m with the signing key sk, obtaining the signature σ. We highlight that in this case the signing algorithm s must be fast and computationally efficient, not involving asymmetric cryptographic primitives, or more generally, modular exponentiations.

The combination of Eq. (1) and Eq. (2) provides the actual signature for the message m, i.e., $< \sigma, pk, \Sigma >$, that has to be transmitted to the client side for the verification phase.

The client verifies pk authenticity by means of Σ, i.e., checking $V_{PK}(pk, \Sigma)$, and subsequently checking the message authenticity by means of $v_{pk}(m, \sigma)$. We stress that the server can split heavy load operations (signing with Eq. (1)) to offline periods and use a fast signing algorithm (s) when it has to deal with a new message. On the other side, the workload at the client side has marginal increase because verification of one-time signatures is also computationally efficient, and the verification algorithm (V) involves a public key exponentiation, which is computationally fast. Figure 4 gives an overview of the online/offline digital signature scheme run by both the client and the server.

5.1. OTS: Lamport's One-Time Signature Scheme

Many different procedures have been proposed in order to implement On/Off signatures, but to the best of our knowledge, the most computationally efficient turns out

[1]We denote a signature as trusted if it can be verified leveraging a PKI. In this article, we always assume the server public/verification key PK as trusted.

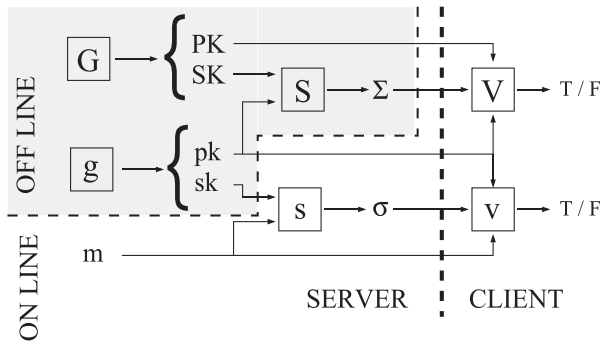

Fig. 4. Online/Offline digital signatures: a general structure.

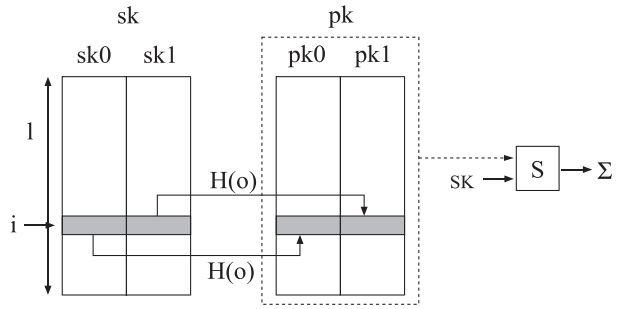

Fig. 5. OTS costruction: offline phase.

to be the *original one time signatures scheme* proposed by Lamport, hereafter OTS. In fact, in such cases the online signing procedure can be implemented with a few system calls: *memcpy*.

We start the analysis of the OTS from the offline phase; see Figure 5 for the details. We assume that the server is already provided with a pair of RSA-based signing/verification keys: *PK* and *SK* in Figure 4. We observe that the offline phase is characterized by three main achievements: sk, pk, and Σ. As will be clear in the following, in order to sign a message of l bits, we need $2l$ tuples of sk and pk but we transmit back to the client only l tuples of each. Figure 5 shows the construction procedure. First, the server generates $2 \cdot l$ tuples of random bits, then *OTS signing keys*, $sk = [sk0, sk1]$. Subsequently, the server computes $pk0$ and $pk1$ from $sk0$ and $sk1$, respectively. $pk0_i = H(sk0_i)$ and $pk1_i = H(sk1_i)$, where $H(\circ)$ is a cryptographic secure hash function such as SHA-1, and $i \in [0, L-1]$ is the index identifying the i^{th} processed tuple. Hereafter we refer to $pk = [pk0, pk1]$ as the *OTS verification keys*. Finally, the server generates the trusted signatures Σ signing the OTS verification keys, $\Sigma = S_{SK}(pk)$, where $S_{SK}(\circ)$ is the RSA signing procedure introduced by Eq. (1).

We observe that as the previous procedure is slow and computationally expensive, in fact, in order to generate l tuples, the server has to compute $2 \cdot l$ hash functions and 1 RSA-based signature. We stress how this procedure can be computed offline or even delegated to other machines, in fact, it is completely independent of the message to be signed.

Recalling Figure 4, we observe that the online signing phase at the server side generates σ from sk and m. Figure 6 shows the details of this procedure. The message m is processed bit-by-bit: without loosing of generality, hereafter we assume the length

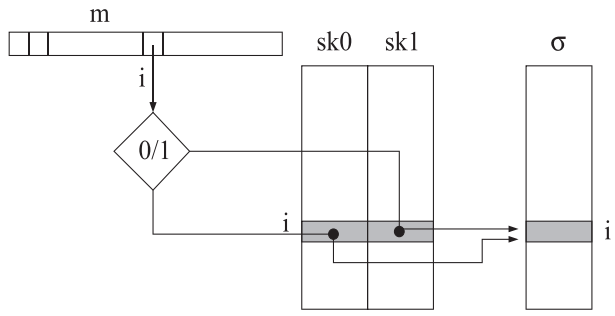

Fig. 6. OTS costruction: online phase.

of message m as l bits. The OTS signature σ is generated from the OTS signing keys sk as a function of the i^{th} bit value of the message m. In more detail, the algorithm set the σ_i value choosing between $sk0_i$ and $sk1_i$ as function of the i^{th} message bit m_i, e.g., if $m_i = 0$ than $\sigma_i = sk0_i$, while if $m_i = 1$ than $\sigma_i = sk1_i$.

We observe that the online signing procedure at the server side is fast and does not involve any cryptographic procedures, i.e., it can be implemented with l system calls, *memcpy*.

5.2. OTS at the Client Side

We recall from Section 5 that the actual signature transmitted from the server to the client in order to authenticate the message m is constituted by $< \overline{PK}, PK, \Sigma, pk, \sigma >$. First, in order to accomplish the authentication procedure, the client must be able to verify the server RSA-based verification key PK by leveraging both the digital signature $\overline{PK}$ and the trusted third party CA. After the reception of $< \overline{PK}, PK, \Sigma, pk, \sigma >$ the client has to perform two main actions: (1) verifying the authenticity of pk by means of Σ, and (2) checking the validity of σ by means of pk.

Recalling that $\Sigma = S_{SK}(pk)$, the client can verify pk authenticity by running $V_{PK}(pk, \Sigma)$ as previously discussed for Figure 3. Now, the client checks the message authenticity by reversing the procedure performed by the server. In particular, the client processes each bit m_i of the message m, and verifies m_i validity by checking that $H(\sigma_i) = pk0_i$ if $m_i = 0$ or $H(\sigma_i) = pk1_i$ if $m_i = 1$.

We observe that the client workload sums up to one RSA-based signature verification and l hash computations.

6. PUTTING EVERYTHING TOGETHER

In this section we combine the concepts previously introduced in order to provide the final design of our key-establishment protocol.

Figure 7 shows the details of the proposed protocol. As in the standard TLS/SSL handshake, the client starts the protocol. Actually, the client is supposed to be provided with a pair of RSA-based public/private keys, hereafter pK, sK respectively; such keys can be generated by the client before running the key-establishment protocol.

The client is also asked to negotiate a few parameters with the server before starting the protocol; nevertheless, in the following, we do not deal with such aspects, and we assume the pair client-server is aware of the protocol and the parameters in order to make it work properly.

The client sends its own public key pK to the server. The server generates the master secret key K_s and encrypts it with the client public key pK, obtaining the message M_s.

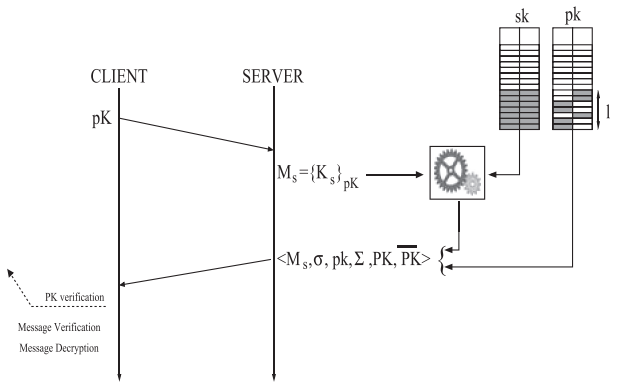

Fig. 7. A sketch of the proposed key-establishment protocol.

Now, the server signs the message $m = H(M_s)$ with the OTS algorithm presented in Section 5.1, generates a new message $< M_s, \sigma, pk, \Sigma, PK, \overline{PK} >$, and sends it back to the client. This step consumes exactly l tuple from each of the precomputed data structures sk and pk.

The communication overhead between the server and the client can be reduced since only half of the tuples of pk need to be transmitted by the server to the client. In fact, recalling that $pk_i = H(sk_i)$ (Figure 5), and only half of the sk_i are selected to build up σ (Figure 6), the server has to send to the client, only the pk_i that do not correspond to the sk_i selected during the online phase, i.e., the others can be computed by the client.

On the other side, the client verifies the server public key PK by means of $\overline{PK}$ and CA. Second, the client verifies the message m by means of the OTS signature (see Section 5.2), and finally, decrypts M_s with sK, obtaining the secret key K_s, $K_s = \{M_s\}_{sK}$. Now, both the client and the server share a secret key and can leverage a symmetric encryption algorithm to secure their subsequent communications.

So far, we did not consider any attacks on the server. Clearly, in the current configuration, the protocol in Figure 7 can be easily exploited by both a reply attack or a denial of service attack. We show how to effectively deal with these attacks in Section 9.

7. SECURITY ANALYSIS

In this section we present the security analysis of the LAKE protocol. We start by recalling the security model developed by Canetti and Krawczyk [2001], hereafter CK model, which so far, is the most comprehensive and generic security model for analyzing authenticated key-establishment protocols. We subsequently provide a formal analysis of the LAKE protocol under the CK model, and finally, prove it as secure under such a model.

7.1. CK Model: Assumptions

The CK model assumes three different entities interacting with each other: the adversary $\mathcal{ADV}$, and the parties P_i and P_j running the protocol π to be proved as secure. The parties interact between them running a session s of the protocol π with the aim of establishing a session key K_s.

A session can be in one of the following states: (1) *incomplete*, when it has not yet produced any outputs or (2) *completed*, when it has returned an output with a nonnull key value. Each session is characterized by an internal state, which comprises the

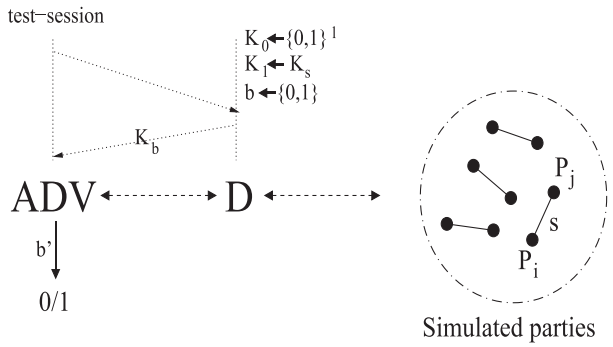

Fig. 8. CK-model: The game $\Gamma_{ADV}(l)$ played by the adversary $\mathcal{ADV}$ against the simulator $\mathcal{D}$.

parties local working space to the session itself. In the following, we refer to the session s, involving the client P_i and the server P_j as (P_i, P_j, s).

$\mathcal{ADV}$ interacts with the parties in three different ways.

— *Activation*. $\mathcal{ADV}$ might activate a party P_i to initiate the protocol π or P_j to answer to an incoming request from P_i.
— *Compromisation*. The model envisages three possible attacks.
  (i) *Corruption.* All the secrets of the target party are disclosed.
  (ii) *Session-state reveal.* This attack can be applied to any incomplete sessions and discloses the local state of the target session. As will be clear in the following, the local state disclosure does not leak any information on the long-term key, i.e., the private key of the party.
  (iii) *Session-key query.* This attack can only be applied to any completed sessions and discloses the session key corresponding to the target session.
— *Test-session*. $\mathcal{ADV}$ might ask for a test-session to a party. The target party answers with a key K_b, with $b \in \{0, 1\}$, where $K_0 \stackrel{\$}{\leftarrow} \{0, 1\}^l$, and $K_1 \leftarrow K_s$. Therefore, the target party randomly chooses the key to be returned to $\mathcal{ADV}$ between the real session key ($K_1 = K_s$) and a random string (K_0).

7.2. CK Model: The Game

The CK model is based on a game $\Gamma_{ADV}(l)$ played by the adversary $\mathcal{ADV}$ against the simulator $\mathcal{D}$ under a security parameter l. $\mathcal{D}$ simulates the interactions between the parties $\{P_1, \ldots, P_N\}$ that run the protocol π to be proved as secure. Figure 8 shows the main entities involved in the game. $\mathcal{D}$ behaves as a proxy between $\mathcal{ADV}$ and the parties. In particular, $\mathcal{D}$ answers transparently to all the requests of $\mathcal{ADV}$ (activations and compromisations), but challenges $\mathcal{ADV}$ when this asks for a test-session. As an example, when $\mathcal{ADV}$ activates the protocol π between P_i and P_j, with $i, j \in [1, \ldots, N]$, $\mathcal{D}$ answers simulating the protocol between the two parties; yet, when $\mathcal{ADV}$ performs a session-state reveal or a session-key query, $\mathcal{D}$ behaves accordingly. Nevertheless, when $\mathcal{ADV}$ performs a test-session on the session s, $\mathcal{D}$ behaves differently: it stores the session key of the party P_i on the temporary variable K_1, $K_1 \leftarrow K_s$, and computes a random key K_0 from a uniform distribution, $K_0 \stackrel{\$}{\leftarrow} \{0, 1\}^l$. Subsequently, it extracts a random value b from a uniform distribution, $b \stackrel{\$}{\leftarrow} \{0, 1\}$, and returns K_b to $\mathcal{ADV}$, accordingly, it returns K_0 (K_1), if $b == 0$ ($b == 1$), respectively. Eventually, $\mathcal{ADV}$ returns b' as its guess for b.

(1) Set-up: Declare the security parameter l and initialize the parties $\{P_1, \ldots, P_N\}$.

(2) $\mathcal{ADV}$ performs one of the following actions:
- Activation:
  - — Activate an action request message to party P_i.
  - — Activate an incoming message from party P_i to party P_j.
- Compromisation
  - — Corrupt a party P_i.
  - — Issue a session-state reveal query to an incomplete session of a party P_i.
  - — Issue a session-key query to a completed but unexpired session of a party P_i.
- Test-session of a completed, unexpired, and unexposed session. A value K_b is returned to $\mathcal{ADV}$.

(3) $\mathcal{ADV}$ might continue with the actions at step (2), but it is not allowed to expose the test-session.

(4) $\mathcal{ADV}$ returns a bit b' as its guess for b. The output of the game is b'.

Fig. 9. The distinguishing game $\Gamma_{ADV}(l)$.

THEOREM 1. *A key-establishment protocol is called session-key secure, hereafter SK-secure, in the CK model if the following conditions hold.*

(*1*) *If two uncorrupted parties complete a session, then they both output the same session key.*

(*2*) *The advantage that $\mathcal{ADV}$ wins the distinguishing game $\Gamma_{ADV}(l)$ at the end of the experiment (see Figure 8) is negligible in the security parameter l.*

$$P(\Gamma_{\mathcal{ADV}}(l) = b) \leq \frac{1}{2} + \epsilon(l), \tag{3}$$

where $\epsilon(l)$ is a negligible function in the security parameter l.

The first condition says that uncorrupted parties that complete the key-agreement protocol should eventually agree on a shared secret. Most importantly, the second condition states that after running Γ_{ADV}, $\mathcal{ADV}$ should not have any advantage on guessing the session key of a target pair, or at least no more than a random guess.

The details of the game $\Gamma_{ADV}(l)$ are shown in Figure 9.

7.3. CK Model: Adversary Goals

The CK model assumes two different types of adversarial behaviors: the *unauthenticated-links model*, hereafter UM, and the *authenticated-links model*, hereafter AM.

— *AM Model*. $\mathcal{ADV}$ is not allowed to generate, inject, modify, replay, and deliver messages except if the message comes from a corrupted party. In this model, the links are authenticated, and therefore, $\mathcal{ADV}$ can only eavesdrop the messages in the network.

— *UM Model*. $\mathcal{ADV}$ can compromise a party, build, and deliver messages. This is a stronger model in which the adversary does not have the restriction of the AM model, i.e., $\mathcal{ADV}$ can forge and inject messages.

Proving a protocol SK-secure directly under the UM model (by means of the $\Gamma_{ADV}(l)$ game, see Figure 9) might be difficult due to the strong adversarial assumptions. However, Canetti and Krawczyk [2001] proved that it is always possible to transform a protocol proven as secure in the AM into a protocol that is secure in the UM. Let us recall the following theorem.

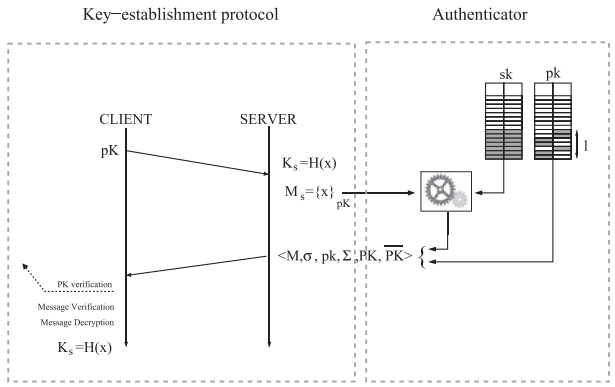

Fig. 10. A sketch of the LAKE protocol using the CK-model.

THEOREM 2. *Let π be a key-establishment protocol that is SK-secure in the AM, and let C be any valid authenticator. Then $\pi' = C(\pi)$ is SK-secure in the UM.*

Therefore, given a key-establishment protocol π, SK-secure in the AM, it is always possible to transform it into a protocol π', which is SK-secure in the UM. This is possible because of the authenticator C, which allows the parties to authenticate the links involved in the communications. Canetti and Krawczyk [2001]. provided three examples of valid authenticators in their seminal paper. Although all of the proposed authenticators can be adopted in our proposal, none are computationally efficient for the server side. Therefore, in the following we split the LAKE protocol into two building blocks: the LKE key-establishment protocol and the authenticator λ_{LAKE}. We start our security analysis by proving LKE as secure in the AM, and subsequently, we show that λ_{LAKE} is a valid authenticator and can be applied to LKE to obtain a SK-secure protocol in the UM, i.e., the LAKE protocol.

7.4. LAKE Components under the CK Model

The CK model involves mainly three steps to design an SK-secure protocol in the UM.

(1) Design a basic key-establishment protocol and prove it SK-secure in the AM.
(2) Design an authenticator and prove that it is valid.
(3) Apply the authenticator to the basic protocol to produce a protocol that is automatically secure in the UM model according to Theorem 2.

Figure 10 shows the LAKE protocol under the CK model. We consider two building blocks: the key-establishment protocol, hereafter LKE, and the authenticator, hereafter λ_{LAKE}. In order to prove the security of the LAKE protocol in the UM, we follow the subsequent steps: first, we prove the LKE protocol as secure in the AM by playing the game $\Gamma^{LKE}_{ADV}(l)$; second, we prove λ_{LAKE} is a valid authenticator, and finally, we prove the security of LAKE in the UM by combining LKE and λ_{LAKE} under Theorem 2.

7.5. Security of the LKE Key-Establishment Protocol

In the following we prove the security of the LKE protocol (the key-establishment protocol in LAKE).

ASSUMPTION 1. *The encryption scheme $\{\circ\}_{pK}$ is robust to the chosen cyphertext attack, namely $\{\circ\}_{pK}$ is CCA-secure.*

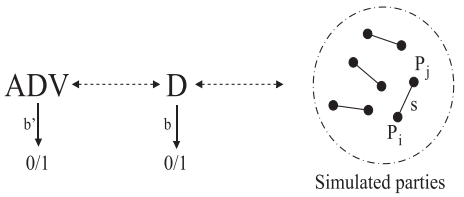

Fig. 11. Entities $\mathcal{ADV}$ and $\mathcal{D}$ and their interaction in the game Γ^{LKE}.

An encryption/decryption system is considered secure in terms of indistinguishability (CCA-secure) if no adversary provided with an encryption of a message randomly chosen from a two-element message space determined by the adversary itself, can identify the message choice with probability significantly better than that of a random guess. Therefore, assuming the adversary knows both the plaintexts $\{x, x'\}$ and the public key pK, he cannot identify $M = \{x\}_{pK}$ from $M' = \{x'\}_{pK}$, provided he is not aware of the private key sK.

THEOREM 3. *Protocol LKE is SK-secure in the AM assuming the encryption scheme* $\{\circ\}_{pK}$ *as CCA-secure.*

PROOF. In order to prove the theorem, we have to show that LKE satisfies both Conditions 1 and 2 of Theorem 1.

Condition 1. It is easy to see that the first condition is satisfied by the LKE protocol, i.e., uncorrupted parties that complete a session output the same session key. If the client and the server complete a session while uncorrupted, then the server must have received the client public key pK, while in turn, the client must have received the encrypted random nonce generated by the server, i.e., $M_s = \{x\}_{pK}$. This is so because an AM adversary, such as $\mathcal{ADV}$, is not allowed to modify or inject messages belonging to uncorrupted parties. Thus, the client first decrypts the received message from the server, $x = \{M_s\}_{sK}$, and second, computes $K_s = H(x)$ and retrieves the session key K_s generated by the server.

Condition 2. We prove the second condition of Theorem 1 by reductio ad absurdum. We assume that there exists an adversary $\mathcal{ADV}$ in the AM able to win the game Γ^{LKE}_{ADV} (see Figure 9) with nonnegligible probability

$$P(\Gamma_{\mathcal{ADV}}(l) = b) = \frac{1}{2} + \overline{\epsilon}(l), \tag{4}$$

where $\overline{\epsilon}(l)$ is a nonnegligible function in the security parameter l. We show that, under such a condition, the encryption scheme is not CCA-secure, thus contradicting the assumption.

We start by defining a game Γ^{LKE} that captures the CCA-security of the encryption function $\{\circ\}_{pK}$. The game Γ^{LKE} is constituted by two entities: the adversary, $\mathcal{ADV}$, and the simulator, $\mathcal{D}$. The adversary invokes the game interactions, while the simulator performs the protocol executions. Figure 11 shows the interactions between the entities. The simulator $\mathcal{D}$ answers to the $\mathcal{ADV}$'s requests and simulates a virtual scenario constituted by several parties running the LKE protocol. Eventually, $\mathcal{ADV}$ outputs the bit b' as its guess for the bit b randomly generated by $\mathcal{D}$. The details of the game Γ^{LKE} are the following.

(1) Set up the security parameter l and initialize parties $\{P_1, \ldots, P_n\}$.

Let $s \xleftarrow{\$} [1,\ldots,S]$ be a generic session that involves the client P_i and the server P_j (as in Figure 11), where S is the upper bound on the number of sessions in any interaction. For each session (P_i,P_j,s), let x_s, K_s, and M_s be the random nonce, the secret key and the encrypted secret key, respectively, as depicted in Figure 10.

(2) Invoke $\mathcal{ADV}$ to interact with parties $\{P_1,\ldots,P_n\}$ running the LKE protocol in the AM, yielding the following.
   — *Message exchanges*. If $\mathcal{ADV}$ activates (P_i,P_j,s), then let P_i send (P_i,s,pK) to P_j. If P_j receives (P_i,s,pK), let P_j send (P_j,s,M_s) to P_i, where $M_s = \{x_s\}_{pK}$ and $x_s \xleftarrow{\$} \{0,1\}^l$, respectively. All the computations and message exchanges are simulated by $\mathcal{D}$.
   — *Party corruption*. If $\mathcal{ADV}$ corrupts party P_i, then $\mathcal{D}$ gives $\mathcal{ADV}$ all information about party P_i.
   — *Session exposure*.[2] If $\mathcal{ADV}$ exposes session (P_i,P_j,s), then $\mathcal{D}$ gives $\mathcal{ADV}$ all information on the session s.
   — *Test-session query*. If $\mathcal{ADV}$ picks (P_i,P_j,s) as the test session, then $\mathcal{D}$ chooses a random bit $b \xleftarrow{\$} \{0,1\}$. If $b == 0$ then $\mathcal{D}$ gives K_s (the actual session key) to $\mathcal{ADV}$, otherwise K'_s, where $K'_s \xleftarrow{\$} \{0,1\}^l$, i.e., a random bit sequence.

(3) $\mathcal{ADV}$ returns the bit b' as its guess for the bit b.

We observe that Γ^{LKE} is a perfect-simulation of the protocol LKE under the UM, as defined by Canetti and Krawczyk [2001]. Yet, we observe that $\mathcal{ADV}$ has the same chances of a random guess to identify the correct b after the test-session query—assuming LKE as a CCA-secure protocol, $P(b = b') = \frac{1}{2}$.

Now, we prove the security of the LKE protocol. As stated before, the strategy relies on assuming $\mathcal{ADV}$ as able to break LKE, and leveraging this in order to set up a CCA-distinguisher, which by assumption is absurd (Assumption 1). Instead of the simulator $\mathcal{D}$, we consider a simulator $\mathcal{D}_\mathcal{A}$, which leverages the power of $\mathcal{ADV}$ (able to break LKE) in order to implement a CCA-distinguisher. $\mathcal{D}_\mathcal{A}$ works as follows.

— Let $\mathcal{D}_\mathcal{A}$ pick a random session $s^* \xleftarrow{\$} [1,\ldots,S]$, which in turn, identifies two parties P_i^* and P_j^*, respectively. Let also, $\mathcal{D}_\mathcal{A}$ generate a pair of private/ public keys, (sK^*,pK^*) respectively, and compute $K^* = H(x^*)$ and $M^* = \{x^*\}_{pK^*}$, where $x^* \xleftarrow{\$} \{0,1\}^l$.
— *Message exchanges*. If $s \neq s^*$ then $\mathcal{D}_\mathcal{A}$ behaves as $\mathcal{D}$, otherwise if $\mathcal{ADV}$ activates (P_i^*,P_j^*,s^*), then let P_i^* send (P_i^*,s^*,pK^*) to P_j^*. If P_j^* receives (P_i^*,s^*,pK^*), let P_j^* send (P_j^*,s^*,M^*) to P_i^*. Therefore, when $\mathcal{ADV}$ instantiates the session s^*, $\mathcal{D}_\mathcal{A}$ injects (transparently to $\mathcal{ADV}$) its own generated crypto material.
— *Party corruption*. If $\mathcal{ADV}$ corrupts party P_j^*, the server in the session s^*, then $\mathcal{D}_\mathcal{A}$ outputs $b \xleftarrow{\$} \{0,1\}$ and aborts the simulation. Otherwise, $\mathcal{D}_\mathcal{A}$ behaves as $\mathcal{D}$.
— *Session exposure*. If $\mathcal{ADV}$ exposes session s^*, then $\mathcal{D}_\mathcal{A}$ outputs $b \xleftarrow{\$} \{0,1\}$ and abort the simulation. Otherwise, $\mathcal{D}_\mathcal{A}$ behaves as $\mathcal{D}$.
— *Test-session query*. If $\mathcal{ADV}$ picks session s^* as the test session, then $\mathcal{D}_\mathcal{A}$ gives K^* to $\mathcal{ADV}$ and returns $b = 0$. Otherwise, $\mathcal{D}_\mathcal{A}$ behaves as $\mathcal{D}$.
— If $\mathcal{ADV}$ halts without choosing a test-session then $\mathcal{D}_\mathcal{A}$ outputs $b \xleftarrow{\$} \{0,1\}$.

[2]In order to ease the exposition, we refer to session exposure as all the session compromisation activities that can be performed by $\mathcal{ADV}$, i.e., session-state reveal and session-key query.

We observe that $\mathcal{D}_{\mathcal{A}}$ is specifically designed to leverage the $\mathcal{ADV}$ power of breaking the LKE protocol. In fact, $\mathcal{D}_{\mathcal{A}}$ behaves as $\mathcal{D}$ for all the sessions, except for the session s^*. For such a session, $\mathcal{D}_{\mathcal{A}}$ transparently feeds $\mathcal{ADV}$ with self-generated crypto-material, i.e., sK^*, pK^*, M^* and K^*, and as will be clear in the following, this turns $\mathcal{D}_{\mathcal{A}}$ into a CCA-distinguisher—which is an absurd.

Now, in order to demonstrate (the absurd) that $\mathcal{D}_{\mathcal{A}}$ is a CCA-distinguisher, we consider the probability that both $\mathcal{ADV}$ and $\mathcal{D}_{\mathcal{A}}$ output the same bit value, showing that this probability is not negligible, $P(b = b') > \frac{1}{2}$. Let E be the event such that $\mathcal{ADV}$ picks the session s^* as the test session, the probability that E occurs can be computed as $P(E) = \frac{1}{S}$. The probability that $\mathcal{ADV}$ and $\mathcal{D}_{\mathcal{A}}$ output the same bit value when E occurs is the same as the probability of $\mathcal{ADV}$ to win the Γ^{LKE} game, i.e., when E occurs, $\mathcal{D}_{\mathcal{A}}$ always returns $b = 0$, while $\mathcal{ADV}$ returns $b' = 0$ if and only if it breaks the LKE protocol, yielding

$$P(b = b'|E) = P(\Gamma^{LKE}_{\mathcal{ADV}}(l) = b).$$

Conversely, the probability that $b = b'$ when E does not occur yields

$$P(b = b'|\overline{E}) = \frac{1}{2}.$$

In fact, if $\mathcal{ADV}$ instantiates any session except s^* the output of $\mathcal{D}_{\mathcal{A}}$ is random. Finally, the unconditioned probability that $b = b'$ can be computed as

$$P(b = b') = P(b = b'|E)P(E) + P(b = b'|\overline{E})P(\overline{E}),$$

which in turn can be expressed as

$$P(b = b') = P(\Gamma^{LKE}_{\mathcal{ADV}}(l) = b)\frac{1}{S} + \frac{1}{2}(1 - \frac{1}{S}),$$

and substituting Eq. (4) in the previous, it yields

$$P(b = b') = \frac{1}{2} + \frac{\overline{\epsilon}(l)}{S},$$

which is not negligible by assumption.

Therefore, we state that, given a CCA-secure encryption scheme, the LKE protocol is proved as secure under the AM in the CK model. □

Remark. The security of protocol LKE assumes that operations related to the computation of session keys are executed in a separate secure module [Tin et al. 2003]. Therefore, only the session key K_s is disclosed by the session exposure and not the long-term secrets such as the private key.

7.6. Validity of the LAKE Authenticator

In this section we analyze the security of the LAKE authenticator, hereafter λ_{LAKE}.

THEOREM 4. *λ_{LAKE} is a valid authenticator if the signature scheme (G, S, V) is forgery proof.*

PROOF. We consider an adversary able to break the λ_{LAKE} authenticator and we prove that if such an adversary exists, then it is a successful signature forger.

We recall the λ_{LAKE} authenticator in Figure 12. Figure 12(a) shows the signature generation: we assume sk, pk, Σ, and PK as available from the offline phase. Moreover, we recall that the online phase sums up to a selection of the elements in sk as a function of the bit values in m. The resulting signature (σ, pk, Σ, PK) is sent to the client for the verification procedure.

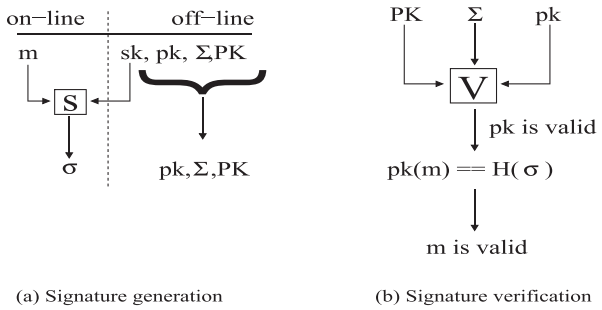

Fig. 12. The λ_{LAKE} authenticator: (a) signature generation and (b) signature verification.

As for the verification (Figure 12(b)), in order to ease the discussion, we refer to the selection of the elements of pk, as functions of the bit values in m, as $pk(m)$, i.e., recalling Figure 5, $pk0_i$ ($pk1_i$) is selected if the i^{th} bit of m is equal to 0 (1). Therefore, the signature verification is constituted by two steps: (1) pK verification and (2) message authentication. We assume the public key PK as trusted, i.e., its authenticity can be verified by means of a trusted third party or it is sent to the client via an out-of-bandwidth trusted channel.

If the adversary can break the λ_{LAKE} signature verification procedure, then the client will accept a message m' as coming from the purported server, while it has been generated by $\mathcal{ADV}$ and sent to the client with the following signature $(\sigma', pk', \Sigma, PK)$, where σ' and pk' have been generated by $\mathcal{ADV}$, while Σ and PK have been retrieved by eavesdropping the client-server communication. Yet, if $\mathcal{ADV}$ breaks λ_{LAKE}, it means that the verification step of pk' is successfully performed by the client, i.e., $V_{PK}(pk', \Sigma) = true$, therefore $\mathcal{ADV}$ has been able to forge the signature Σ by $\Sigma = S_{SK'}(pk')$, where SK' is the $\mathcal{ADV}$'s signature key. This is a contradiction with the assumption that $\Sigma = S_{SK}(pk)$. □

7.7. Security of LAKE in the UM

Theorem 2 from Canetti and Krawczyk [2001] provides a modular methodology in order to prove an authenticated key-establishment as secure. This approach involves proving the security of the protocol on a simplified scenario, i.e., a game Γ_{ADV} played by a weak adversary (AM model), and subsequently extending it by means of a valid authenticator (λ_{LAKE}).

We proved π = LKE as SK-secure in AM (Section 7.5) and we validated the authenticator $C = \lambda_{LAKE}$ (Section 7.6). Now, recalling Theorem 2, we state that protocol π' = LAKE is secure under the UM by combining LKE with λ_{LAKE}, i.e., $\pi' = C(\pi)$.

8. PERFORMANCE EVALUATION

8.1. The Security Parameters *l* and *L*

The security of our protocol depends on the parameters l and L. Table III shows a comparison in terms of security among different key lengths and crypto algorithms, e.g., the RSA-based encryption with a key length of 1024 bits provides almost the same security as a symmetric encryption algorithm that adopts a key length of 80 bits [Orman and Hoffman 2004]. Therefore, in the following, we assume $l \in \{56, 80, 104, 128, 144\}$ with the corresponding RSA key length, $L \in \{512, 1024, 2048, 3072, 4096\}$, respectively. Recalling Figure 7, we assume the secret key K_s has a fixed length of l bits, while the encrypted message M_s is in turn L bit long. Nevertheless, a one-time signature is computed on an l bit long message, i.e., the first l bits of $m = H(M_s)$. Therefore, in order

Table III. Security Equivalence between Asymmetric and Symmetric Crypto (Key-Lengths)

RSA-based crypto (L)	Symmetric crypto (l)
512	56
1024	80
2048	104
3072	128
4096	144

Table IV. Client-Server Workloads for the Proposed Key-Establishment Protocol

	CLIENT side	SERVER side	
		Online	Offline
Hash Computations	l	1	$2 \cdot l$
RSA Signing	0	0	1
RSA Verifying	1	0	0
RSA Encryption	0	1	0
RSA Decryption	1	0	0

Table V. Client-Server Workloads for the TLS/SSL Standard Handshake

	CLIENT side	SERVER side
Hash Computations	1	1
RSA Signing	0	0
RSA Verifying	1	0
RSA Encryption	1	0
RSA Decryption	0	1

to sign a message m, each of sk and pk (Figure 5) needs l tuples of length l bits each one, i.e., for the pk_i computations we again consider only the first l bits of the SHA-1 output.

8.2. Qualitative Analysis

Table IV shows a summary of the crypto functions run by both the client and the server in order to establish the secret key K_s encrypted with an RSA-based public key procedure that generates a message M_s of L bits. We want to stress that the offline phase at the server side can be computed by the server itself during periods of low CPU utilization or even by other machines. Yet we observe that the server undergoes only one RSA-based public key encryption during its online phase while all the workload is moved to the offline phase. On the other side, the client workload sums up to l hash computations, 1 RSA verification, and finally 1 RSA decryption.

In the following, we provide a qualitative comparison between the proposed approach and the standard TLS/SSL handshake introduced in Section 3.3.

We start our analysis from the client side. The client has to accept an increasing computational workload moving from the TLS/SSL handshake to our protocol, in fact, the number of hash computations passes from 1 to l. Nevertheless, the client saves an RSA encryption but has to perform an RSA decryption, which is generally more computationally expensive.

At the server side (online phase), we observe a severe improvement of the performance between the TLS/SSL handshake and our proposed protocol. The server saves the RSA decryption and it needs only one RSA encryption, which is computationally less expensive.

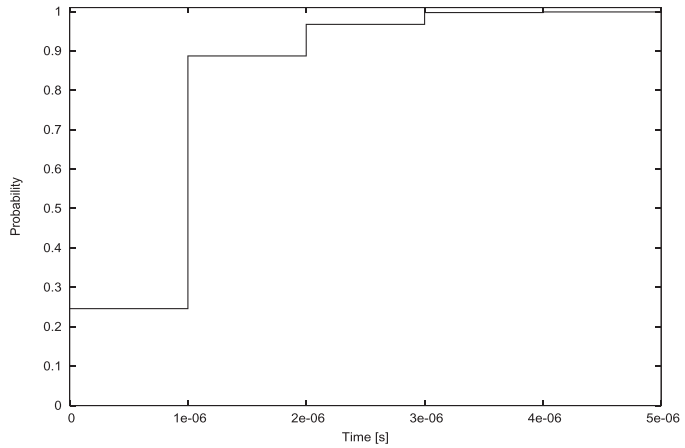

Fig. 13. Probability distribution function associated to the SHA-1 execution time.

8.3. A Real Testbed

In order to test the real performance of our protocol, we implemented it by means of the OpenSSL library (version 0.9.8) [OpenSSL 2012]. We developed a client-server pair that runs the protocol proposed in Section 6. The software [Oligeri 2012] is constituted by three main files: the *server*, which waits for a new incoming connection, the *client*, which initiates a new connection, and finally, the *generator*, which generates the OTS tables shown in Figure 5. As stated before, the *generator* is run by the server—or by other machines—asynchronously with respect to the new client requests.

All the results presented in this work have been generated running the *server* on a Intel Core i3 CPU 540@3.07GHz with 8GB of RAM. We performed many client-server connections between two computers on the same local network, and collected statistics stop-watching the code of the critical parts, i.e., measuring the time spent by the CPU in all the cryptographic functions. We considered four RSA key lengths, 512, 1024, 2048, and 4096, respectively, and we adopted secret keys K_s of length equal to 56, 80, 104, 128, and 144 bits, respectively.

We start our analysis by measuring the SHA-1 execution time, hereafter T_R. Figure 13 shows the cumulative distribution function associated to the SHA-1 computation time. We observe how in our case the execution of the SHA-1 algorithm (provided by the OpenSSL library) needs less than 2μs in 96% of the cases. From now on, we consider $T_R = 2\mu s$ as the *reference unit time*, and all the subsequent measures will be provided with respect to it.

8.4. Server Side Performance with our Protocol

Recalling Figure 7, we observe how the server side workload sums up to one RSA-based encryption and one OTS signing procedure during the online phase. Figure 14 shows the time spent by the CPU in order to encrypt the secret key K_s with the RSA-based encryption algorithm, $M_s = \{K_s\}_{pK}$. The measures have been collected with varying RSA key lengths: 512, 1024, 2048, 3072, and 4096; while, the execution time is expressed as function of T_R, i.e., the reference unit time needed to execute the SHA-1 procedure. The error bars in Figure 14 show the quantiles 5, 50, and 95 associated to the time needed by the CPU in order to perform 100 public key encryptions; we observe how the median values are closer to the minimum values: we attribute this phenomena to the CPU interrupts that may delay the computations.

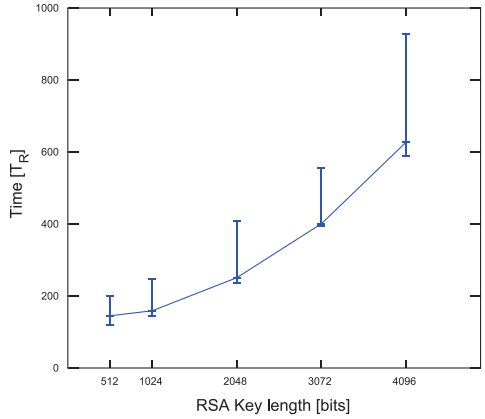

Fig. 14. RSA encryption time at server side.

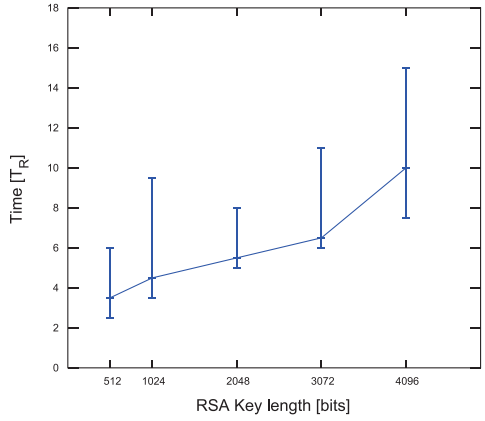

Fig. 15. OTS signature computation time at server side.

Recalling Figure 6, the OTS signing procedure is performed by means of a sequence of system calls *memcpy*, and therefore its overhead depends only on the amount of memory to copy. The length of the message $m = H(M_s)$ to be signed is fixed to l bits, therefore, we estimated the OTS computation time as a function of the RSA key length. Error bars in Figure 15 show the quantiles 5, 50, and 95 associated to the CPU time of 100 OTS signing procedures varying the length of the RSA key length. As for the results presented in Figure 14, we attribute the heavy tails (delays) to the CPU interrupts that may delay the execution of the procedure. Yet, we highlight how the CPU time is almost linear with respect to the key length. As expected, the time needed by the system call *memcpy* is a linear function of the number of bytes to copy.

Finally, we sum up both of the previous computational workloads. Figure 16 shows the stacked histograms of the median time spent by the CPU in both the RSA encryption (green) and OTS signing procedures (blue). First, we highlight how the time spent by the CPU in the signing procedures is negligible with respect to the RSA encryptions for all the cases. Moreover, we observe that, using our protocol, the overall online computational workload sums up to less than 160 SHA-1 computations for a standard RSA-1024 key length, and increases to about 650 SHA-1 computations for the strongest RSA-4096 key.

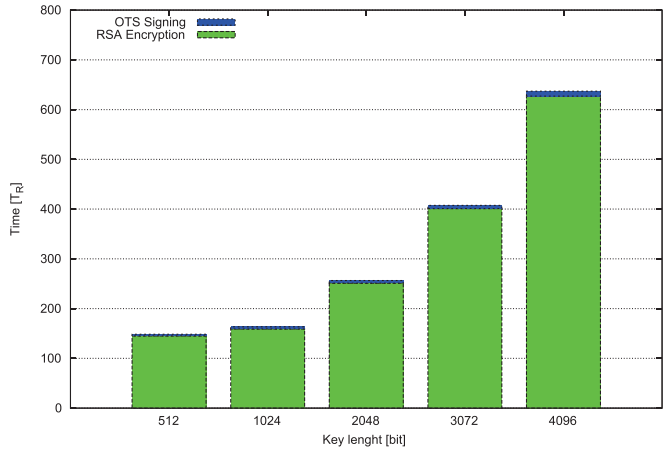

Fig. 16. Our proposed protocol: Total computational workload.

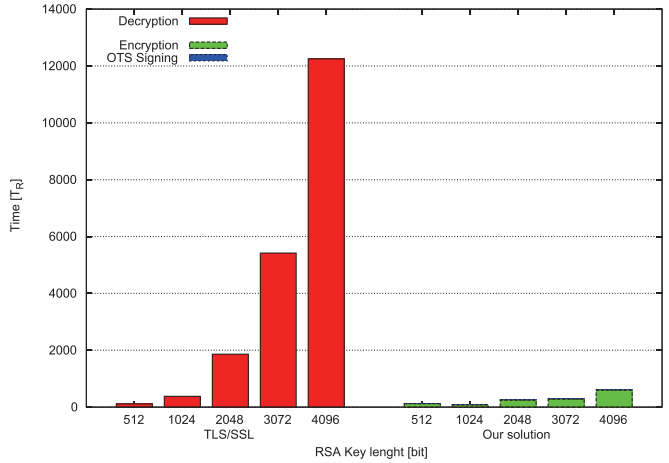

Fig. 17. TLS/SSL VS Our proposed protocol: Total computational workload.

8.5. Server Side Performance with TLS/SSL Handshake

In the following we analyze the computational workload of the standard TLS/SSL handshake. In Table V, we observe that a server running the TLS/SSL handshake undergoes only one heavy RSA procedure: the RSA decryption of the message sent by the client—$\{\{x\}_{PK}\}_{SK}$ in Figure 2. The red histograms in Figure 17 show the median time spent by the CPU during the RSA decryption needed to perform the TLS/SSL handshake. On the other side, the stacked histograms (blue and green) illustrate the performance of our proposed solution (Figure 16).

We observe that the server workload is severely reduced by adopting our protocol with respect to the standard TLS/SSL handshake. In fact, the workload of the later is always heavier when the key size is larger than 512 bits. In particular, we highlight that adopting the standard RSA-1024 key length, our protocol is about 4.4 times faster, but the gain increases when the key length is larger—our protocol is about 7.3, 18.6, and 20.1 times faster when adopting RSA-2048, RSA-3072, and RSA-4096, respectively.

Table VI. Comparison with Previous Solutions

	Castelluccia	Bicakci	Our Theo.	Our Exp.
RSA-512	-	8.5	10	0.9
RSA-1024	11.33	16.2	18	4.4
RSA-2048	-	33.2	33	7.3
RSA-3072	-	-	43.5	18.7
RSA-4096	19.12	64.1	61	20.1

8.6. Comparison with Other Solutions

In this section we compare the performance of our protocol with that obtained by Castelluccia et al. [2006] and Bicakci et al. [2006].

First, we observe that our performance results have been obtained running a real server and a real client on a local network, while performance results by others have been obtained by running the *speed* test provided by the OpenSSL library. Second, the results from Bicakci et al. [2006] are only related to signing/verification RSA procedures and not to encryption/decryption procedures; in fact the authors assumed signing and verification have the same workloads for decryption and encryption.

In order to compare performance, we summarize the workloads that the server must undergo in the various cases. In the standard TLS/SSL key-establishment, the server has to perform only one RSA decryption (see Figure 2), while in the solutions of both ourselves and Bicakci et al. [2006], the server undergoes one RSA encryption and the OTS signing procedure.

Let us introduce the *speedup* factor S, which takes into account the ratio between the RSA decryption and the encryption computation times, such as

$$S = \frac{\text{T(RSA-Dec.)}}{\text{T(RSA-Enc.)} + \text{T(OTS Sign.)}}.$$

S represents how many times our solution is faster that the standard TLS/SSL key-establishment protocol.

A similar analysis has been performed in both Castelluccia et al. [2006] and Bicakci et al. [2006], therefore all the solutions can be compared with respect to the workload of the TLS/SSL handshake. Table VI summarizes the performance of all the proposed protocols. The Bicakci et al. [2006] solution outperforms the Castelluccia et al. [2006] solution for both the RSA-1024 and RSA-2048 configurations. As stated before, such solutions have been tested in a "synthetic environment", i.e., *openssl speed* is an infinite loop of RSA calls, and therefore, in the last two columns of Table VI, we report the performance of our solution computed in two different ways: *Our Theo.*, which comes from *openssl speed*, and *Our Exp.*, which is related to the real measures presented in Section 8.4. First, we observe that using *openssl speed*, our results confirm the extreme performance gain of the OTS-based key-establishment previously introduced by Bicakci et al. [2006]. Second, we observe how *openssl speed* can only be considered as a theoretical upper bound for the RSA procedures performance, in fact, the performance of a real test-bed is severely reduced, although our solution guarantees a performance speed-up with respect to the standard TLS/SSL handshake in a range between 4.4 and 20.1 as a function of the RSA key length.

8.7. Communication Overhead

In this section we analyze the communication overhead introduced by our solution. Recalling Section 6, the server has to send to the client a message carrying the following information: (1) the encrypted secret key k as M, (2) the OTS signature σ, (3) the OTS

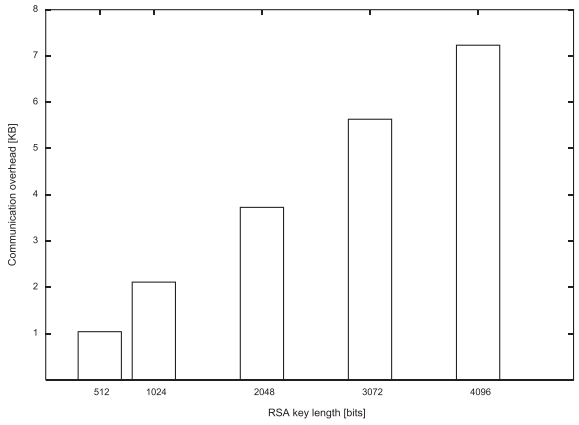

Fig. 18. Communication overhead of our solution as a function of the RSA key length.

verification keys pk, (4) the trusted signature Σ, (5) its own public key PK, and finally, the digital signature $\overline{PK}$. The bandwidth overhead (in bytes) sums up to

$$\frac{L}{8} + l \cdot \frac{l}{8} + l \cdot \frac{l}{8} + \frac{L}{8} + \frac{L}{8} + \frac{L}{8} = \frac{1}{2}L + \frac{1}{4}l^2.$$

Recalling Table III, Figure 18 shows the communication overhead introduced by our solution as a function of the RSA key length. Although we observe that the communication overhead is high, i.e., about 1KB, 2.1KB, 3.7KB, 5.6KB, and 7.2KB, for RSA key lengths equal to 512, 1024, 2048, 3072, and 4096 bits, respectively; we want to stress that, to the best of our knowledge, our solution provides the lowest computational workload at the server side.

9. AVOIDING REPLY ATTACKS AND DENIAL OF SERVICE

The protocol presented in Section 6 can be easily exploited by either a reply attack or a Denial of Service attack (DoS).

A reply attack can be mounted by a malicious third party by repeating or delaying the first communication between the client and server (see Figure 7). We suggest a solution similar to the original TLS/SSL presented in Figure 2: the client generates r_c and sends it to the server together with pK. In turn, the server generates r_s, x and calculates M_s as $k = H(x, r_c, r_s)$, and finally, it encrypts x to obtain M_s. Together with M_s, it also sends r_s to the client (in plaintext). The client decrypts M_s, obtains x, and gets K_s by calculating $H(x, r_c, r_s)$.

As for the DoS attacks, our proposed protocol (even without client puzzles) is in a better position than the standard TLS/SSL handshake to deal with DoS and even dDoS attacks, since it requires less computational load on the server side. An attacker needs to send more bogus requests to bring the server down because the server can now respond to each of client requests by spending less of its computational resources. Nevertheless, we observe that the server answers each new client with a public key encryption and this can be leveraged by malicious clients in order to waste the server's resources. In order to avoid the previous issue, we propose adopting *client puzzles*; [Dean and Stubblefield 2001; Juels and Brainard 1999]. When a new client wants to establish a new secret key with the server, it is requested to solve a computationally expensive puzzle, which prevents malicious clients from sending an excessive number of bogus requests to the server.

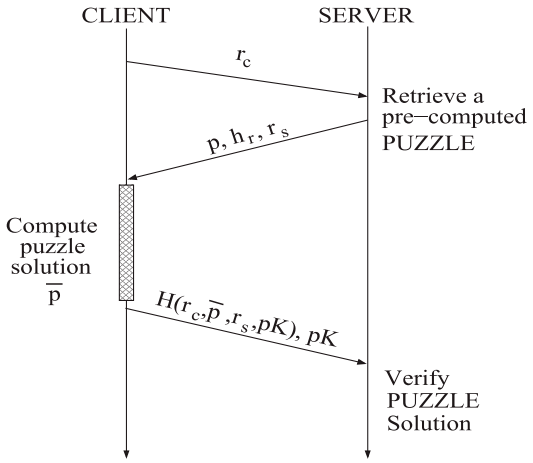

Fig. 19. Avoiding DoS/dDoS with puzzles: the server asks the client for a puzzle solution before initiating the key-establishing protocol.

Therefore, before initiating the real key-establishment protocol, the client-server pair exchanges three messages. Figure 19 shows the messages exchanged in order to negotiate the puzzle: the client initiates the communication sending to the server a random nonce r_c. The server replies with a random nonce r_s and the puzzle (p, h_r).

The puzzle can be precomputed offline as follows: at the server, a random string r of R bits, $r = [r_0, \ldots, r_{R-1}]$, decides the puzzle complexity p_c, and finally, sets to zero the first p_c bits of the random string r obtaining p: $p = [0_0, \ldots, 0_{p_c}, r_{p_{c+1}}, \ldots, r_{R-1}]$. Finally, the server performs a hash computation of r: $h_r = H(r)$.

Now, the client is asked to solve the puzzle, that is, to find a preimage of the received hash h_r leveraging the knowledge of p. Therefore, the client starts an exhaustive search on a space of cardinality 2^{p_c} and will find a solution after an average trial of $2^{p_c}/2$. After finding the solution $\overline{p}$, the client sends back to the server $H(r_c, \overline{p}, r_s, pK)$ and its public key pK. We observe that the puzzle solution $\overline{p}$ is never disclosed on the channel, and therefore, only the client that has solved the puzzle can proceed with the LAKE protocol. In turn, the server verifies the solution by computing only one hash function.

The proposed puzzle allows the server to dynamically choose the complexity. Figure 20 shows the time required for the client in order to find the solution as a function of the puzzle complexity p_c. As in the previous figures, the time refers to the SHA-1 execution time, and each of the error bars represents the quantile 5, 50, and 95, of 100 executions on our reference machine. Finally, we observe that the puzzle solution time is exponential, as expected.

10. CONCLUSIONS

This work proposed a new server-side authenticated key-establishment protocol, which minimizes the server computational workload.

We adopted on/off signatures in order to reduce the online computational workload of the server. In fact, server workload is shifted partially to the client and partially to the offline phases, while the online server computation sums up to only one RSA-based encryption.

To the best of our knowledge LAKE represents the most computationally efficient solution for the server side; in fact, we proved that the server can, theoretically, perform

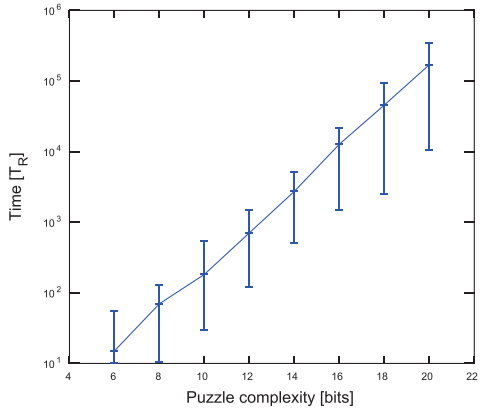

Fig. 20. Time to solve the puzzle as a function of its complexity.

an RSA-2048 based handshake 33 times faster than the standard TLS/SSL handshake (using the same RSA-2048 procedures).

Finally, we have shown how to integrate the LAKE protocol with client-puzzles in order to make it robust to DoS attacks.

Future works are focused on the optimization of the communication overhead of the current protocol.

REFERENCES

Apostolopoulos, G., Peris, V., Pradhan, P., and Saha, D. 2000. Securing electronic commerce: Reducing the SSL overhead. *IEEE Netw. 14*, 4, 8–16.

Bicakci, K., Crispo, B., and Tanenbaum, A. S. 2006. Reverse SSL: Improved server performance and DOS resistance for SSL handshakes. IACR Cryptology ePrint Archive, 212.

Boneh, D., Lynn, B., and Shacham, H. 2001. Short signatures from the Weil pairing. In *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology (ASIACRYPT)*. 514–532.

Canetti, R. and Krawczyk, H. 2001. Analysis of key-exchange protocols and their use for building secure channels. In *Proceedings of EUROCRYPT*. 453–474.

Castelluccia, C., Mykletun, E., and Tsudi K, G. 2006. Improving secure server performance by rebalancing SSL/TLS handshakes. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 26–34.

Chou, W. 2002. Inside SSL: Accelerating secure transactions. *IT Profess. 4*, 5, 37–41.

Coarfa, C., Druschel, P., and Wallach, D. S. 2006. Performance analysis of TLS Web servers. *ACM Trans. Comput. Syst. 24,* 1, 39–69.

Dean, D. and Stubblefield, A. 2001. Using client puzzles to protect TLS. In *Proceedings of the 10th Conference on USENIX Security Symposium (SSYM).* Vol. 10.

DES. 1977. Data encryption standard. In *FIPS PUB 46, Federal Information Processing Standards Publication*. 46–2.

Dierks, T. and Allen, C. 1999. The TLS Protocol version 1.0.

Eastlake 3rd, D. and Jones, P. 2001. US Secure Hash Algorithm 1 (SHA 1), RFC 3174.

Even, S., Goldreich, O., and Micali, S. 1989. On-line/off-line digital signatures. In *Proceedings of CRYPTO.* 263–277.

Fiat, A. 1997. Batch RSA. *J. Crypto. 10,* 2, 75–88.

Guo, F. and Mu, Y. 2008. Optimal online/offline signature: How to sign a message without online computation. In *Proceedings of ProvSec*. 98–111.

Juels, A. and Brainard, J. G. 1999. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS).*

Kant, K., Iyer, R., and Mohapatra, P. 2000. Architectural impact of secure socket layer on Internet servers. In *Proceedings of the International Conference on Computer Design*. 7–14.

Krawczyk, H. and Rabin, T. 2000. Chameleon signatures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

Lamport, L. 1979. Constructing digital signatures from a one-way function. Tech. rep., SRI International Computer Science Laboratory.

Liu, J. K., Baek, J., Zhou, J., Yang, Y., and Wong, J. W. 2010. Efficient online/offline identity-based signature for wireless sensor network. *Int. J. Inf. Secur. 9*, 4, 287–296.

Merkle, R. C. 1987. A digital signature based on a conventional encryption function. In *Proceedings of CRYPTO*. 369–378.

Ming, Y. and Wang, Y. 2010. Improved identity based online/offline signature scheme. In *Proceedings of 7th the International Conference on Ubiquitous Intelligence Computing and Autonomic Trusted Computing (UIC/ATC)*. 126–131.

Oligeri, G. 2012. Server-side authenticated key-establishment. http://gabriele.disi.unitn.it/sw/rhs.tgz.

OpenSSL. 2012. Cryptography and SSL/TLS toolkit. www.openssl.org.

Orman, H. and Hoffman, P. 2004. Determining strengths for public keys used for exchanging symmetric keys. RFC 3766 Best Current Practice.

Potlapally, N. R., Ravi, S., Raghunathan, A., and Jha, N. K. 2006. A study of the energy consumption characteristics of cryptographic algorithms and security protocols. *IEEE Trans. Mobile Comput. 5*, 2, 128–143.

Qing, L. and Yaping, L. 2009. Analysis and comparison of several algorithms in SSL/TLS handshake protocol. In *Proceedings of the International Conference on Information Technology and Computer Science (ITCS)*. 613–617.

Rabin, M. O. 1978. Digital signatures. In *Foundations of Secure Computation*, Academic Press, 155–168.

Rescorla, E. 2000. http over TLS. RFC 2818.

Rivest, R. L., Shamir, A., and Adleman, L. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM 21*, 120–126.

Romanosky, S., Hoffman, D., and Acquisti, A. 2011. Empirical analysis of data breach litigation. In *Proceedings of ICIS*.

Schaad, J. and Housley, R. 2002. Advanced Encryption Standard (AES) key wrap algorithm. RFC 3394.

Shacham, H. and Boneh, D. 2001. Improving SSL handshake performance via batching. In *Proceedings of the Conference on Topics in Cryptology: The Cryptographer's Track at RSA (CT-RSA)*. 28–43.

Shamir, A. and Tauman, Y. 2001. Improved online/offline signature schemes. In *Proceedings of the 21st Annual International Cryptology Conference - Advances in Cryptology (CRYPTO)*. 355–367.

Shen, C., Nahum, E., Schulzrinne, H., and Wright, C. P. 2012. The impact of TLS on SIP server performance: Measurement and modeling. *IEEE/ACM Trans. Netw. 20*, 4, 1217–1230.

Shin, Y., Gupta, M., and Myers, S. 2009. A study of the performance of SSL on PDAs. In *Proceedings of the 28th IEEE International Conference on Computer Communications Workshops (INFOCOM)*. 1–6.

Thiruneelakandan, A. and Thirumurugan, T. 2011. An approach towards improved cyber security by hardware acceleration of OpenSSL cryptographic functions. In *Proceedings of the International Conference on Electronics, Communication and Computing Technologies (ICECCT)*. 13–16.

Tin, Y. S. T., Boyd, C., and Nieto, J. M. G. 2003. Provably secure mobile key exchange: Applying the Canetti-Krawczyk approach. In *Proceedings of the 8th Australasian Conference on Information Security and Privacy (ACISP)*. 166–179.

Yao, A.-C. and Zhao, Y. 2013. Online/offline signatures for low-power devices. *IEEE Trans. Inf. Forensics Security 8*, 2, 283–294.

Zhao, L., Iyer, R., Makineni, S., and Bhuyan, L. 2005. Anatomy and performance of SSL processing. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 197–206.

Received May 2012; revised March 2013; accepted July 2013