Two More Efficient Variants of the J-PAKE Protocol

Jean Lancrenon, Marjan Škrobot and Qiang Tang

SnT, University of Luxembourg
{jean.lancrenon, marjan.skrobot, qiang.tang}@uni.lu

April 14, 2016

Abstract

Recently, the password-authenticated key exchange protocol J-PAKE of Hao and Ryan (Workshop on Security Protocols 2008) was formally proven secure in the algebraic adversary model by Abdalla et al. (IEEE S&P 2015). In this paper, we propose and examine two variants of J-PAKE - which we call RO-J-PAKE and CRS-J-PAKE - that each makes the use of two less zero-knowledge proofs than the original protocol. We show that they are provably secure following a similar strategy to that of Abdalla et al. We also study their efficiency as compared to J-PAKE's, also taking into account how the groups are chosen. Namely, we treat the cases of subgroups of finite fields and elliptic curves. Our work reveals that, for subgroups of finite fields, CRS-J-PAKE is indeed more efficient than J-PAKE, while RO-J-PAKE is much less efficient. On the other hand, when instantiated with elliptic curves, both RO-J-PAKE and CRS-J-PAKE are more efficient than J-PAKE, with CRS-J-PAKE being the best of the three. We illustrate this experimentally, making use of recent research by Brier et al. (CRYPTO 2010). Regardless of implementation, we note that RO-J-PAKE enjoys a looser security reduction than both J-PAKE and CRS-J-PAKE. CRS-J-PAKE has the tightest security proof, but relies on an additional trust assumption at setup time. We believe our results can be useful to anyone interested in implementing J-PAKE, as perhaps either of these two new protocols may also be options, depending on the deployment context.

1 Introduction

The objective of *Password-Authenticated Key Exchange* (PAKE) is to allow secure authenticated communication over insecure networks between two or more parties who only share a low-entropy password. Many different protocols have been proposed in the literature to accomplish this. Among them, the J-PAKE protocol [19] has been implemented due to its patent-free nature.

J-PAKE is quite unique because it integrates Non-Interactive Zero-Knowledge proofs of knowledge (NIZKs in the rest of the paper) - specifically, Schnorr proofs of knowledge [31] - effectively into its design. However, the presence of these proofs is actually one of the main arguments of J-PAKE's detractors: Indeed, they add more exponentiations to a protocol that already contains many. A question that can be asked therefore is whether variants of J-PAKE using less proofs of knowledge can be found, and how they compare in terms of efficiency to the original protocol.

1.1   Our Contribution

We answer these questions by exhibiting two new protocols - which we call RO-J-PAKE and CRS-J-PAKE - that are very similar to J-PAKE, but each use two less zero-knowledge proofs. We explicitly prove the security of RO-J-PAKE, following a similar strategy to that of Abdalla et al. in their recent analysis of J-PAKE [6], and show how the proof can be adapted to the case of CRS-J-PAKE. We also provide a more refined analysis of these protocols' efficiency relative to J-PAKE's. We do this by explicitly examining costs depending on which groups are used to deploy the protocol. This is especially important for RO-J-PAKE, since it requires hashing into the group in question. Indeed, while on paper, this appears to have no importance, in practice it requires some attention. We treat the cases of Elliptic Curve (EC) groups and Subgroups of Finite Fields (SFFs), since all three protocols require the Decisional Diffie-Hellman (DDH) assumption to hold. In more detail, our findings are as follows.

• **In terms of provable security:** RO-J-PAKE and CRS-J-PAKE are asymptotically as secure as J-PAKE, and against the same kind of adversaries, namely, algebraic adversaries. However, RO-J-PAKE enjoys a looser security proof than J-PAKE and CRS-J-PAKE, essentially because of the addition of a random oracle. CRS-J-PAKE has the tightest proof of the three protocols. See the theorem bounds in Section 4.

• **In terms of computational and communication efficiency:** The apparent computational gain in efficiency that RO-J-PAKE and CRS-J-PAKE enjoy due to their having two less zero-knowledge proofs than J-PAKE can be summarized as follows:

  • When all three protocols are instantiated with ECs, CRS-J-PAKE and RO-J-PAKE cost a total of about 8 group-sized exponentiations less than J-PAKE. CRS-J-PAKE has a slight edge over RO-J-PAKE, because the latter requires hashing into an EC group. However, experimental results (see Section 2.4) using recent research by Brier et al. [14] shows that this edge can be practically ignored.

  • When all three protocols are instantiated with SFFs, CRS-J-PAKE takes 8 group-sized exponentiations less than J-PAKE, but RO-J-PAKE suffers from two additional exponentiations of size comparable to that of the base field's prime - which is typically way larger than the actual group - thus making it much less efficient than J-PAKE in practice, see Table 2. This is also due to the need to hash into a SFF. Thus, unless an efficient hashing method is devised, this instantiation of RO-J-PAKE may only have theoretical interest.

  • Regardless of the group instantiation, both RO-J-PAKE and CRS-J-PAKE are more efficient than J-PAKE in terms of communication, as they both send four less group elements and two less scalars than J-PAKE does.

RO-J-PAKE and CRS-J-PAKE have a few other (dis)advantages related to their deployability, and that are worth mentioning. See Section 2.4 for more details.

1.2   Related Work

PAKE in general has been very heavily studied in the past twenty years. We briefly indicate some landmark papers here, and refer to Pointcheval's survey [30] for more complete references. PAKE was introduced by Bellovin and Meritt in [11]. Their EKE protocol was the first of its kind. It was later followed by Jablon's SPEKE protocol [21]. The first viable formal security

models for PAKE appeared in [9] and [13]. A year later, Katz et al. [23] demonstrated that PAKE could be practically realized without random oracles, but at the expense of assuming a Common Reference String (CRS) to be in place. Meanwhile, Goldreich et al. [16] showed that PAKE could be realized in a reasonable security model, solely based on general complexity assumptions, and without any form of trusted setup. Finally, Canetti et al. introduced universally composable PAKE in [15].

Some other work has been devoted to making PAKE more practical for deployment. For instance, in [28] MacKenzie has revisited the PAK protocol [27], showing how to optimize the underlying protocols with EC and SFF implementations. The use of short exponents has also been considered, see [29]. Yet another line of research involved determining the lowest communication costs for standard-model-secure, CRS-based PAKE, see [24, 22]. More recently, work by Abdalla et al. [7] has shown that the computation costs (in terms of number of exponentiations) of many of these protocols can be diminished as well.

The work most relevant to ours is that by Hao and Ryan [19] introducing J-PAKE. The protocol has been deployed in several commercial products and software libraries (e.g. in Firefox sync [2] (later discontinued), OpenSSL [3], PaleMoon [4], and the Thread network protocol [5] (EC version)), mainly because of its simplicity and patent-free nature, but a formal analysis of its security had remained elusive until the work of Abdalla et al. in [6]. Our work is heavily inspired by theirs.

1.3   Organization

The rest of the paper is organized as follows. Section 2 describes our new protocols, and contains a detailed analysis of their efficiency when deployed with EC and SFF. Then, in Section 3, we review the PAKE security model from [8], which is used to prove our protocols' security in Section 4. Finally, we conclude the paper in Section 5.

2   The RO-J-PAKE and CRS-J-PAKE Protocols

In this section we describe the RO-J-PAKE and CRS-J-PAKE protocols, which are presented in Fig. 1 and Fig. 2, respectively. These two protocols can be seen as more efficient variants of J-PAKE protocol from [19]. In addition, we present the practical considerations when these protocols are deployed.

2.1   Notation

For a given security parameter k, let $\mathbb{G}$ be a finite multiplicative group[1] of prime order q, such that $|q| := k$. Being the strongest assumption necessary, we will assume the Decisional Square Diffie Hellman (DSDH, see paragraph 3.2) holds over $\mathbb{G}$. Let H_0 be a full-domain hash mapping $\{0,1\}^*$ to $\mathbb{G}$. H_1 is a hash function from $\{0,1\}^*$ to $\{0,1\}^k$. A function f is used to ensure that both parties sort values identically. This can be done in various ways (e.g. using max or min functions). Let $a \leftarrow A$ denote selecting a uniformly at random from A.

---

[1]As previously mentioned, the group of interest is either a SFF or EC group. Throughout this paper, protocols will be presented multiplicatively.

2.2   The RO-J-PAKE Protocol

As described in [19] and analyzed in [6], the original J-PAKE protocol (see Appendix A) consists of two message rounds. In the first round, each party generates two random group elements and sends them together with corresponding NIZK proofs of the chosen exponents. A client receives X_3 and X_4 values and computes $\alpha := (X_1 X_3 X_4)^{x_2 pw}$, while a server receives X_1 and X_2 values and computes $\beta := (X_1 X_2 X_3)^{x_4 pw}$. In the second round, the client and the server exchange these α and β values, again with corresponding NIZK proofs. In order to compute the shared secret, both parties first cancel the $g^{x_2 x_4 pw}$ factor from the received value, and then exponentiate what is left to either x_2 (client) or x_4 (server). If everything goes according to the protocol's specification, both parties end up with $K := (X_1 X_3)^{x_2 x_4 pw}$.

We observed that the exponents x_1 and x_3 are never explicitly used to compute α, β, or K. Parties only need to use the X_1 and X_3 values to generate what can be considered as a random base $T_K = g^{(x_1 + x_3)}$ for a Diffie-Hellman (DH) transform[2]. Our idea is to exploit this fact and change the protocol such that the number of NIZK proofs in protocol can be reduced. However, as in the proof of the original J-PAKE (see [6]), we still need to know the discrete logs of X_1 and X_3 for the reduction to work (i.e. in order to simulate the protocol in a sound way). A solution for this is to employ a random oracle taking as input fresh messages from each party to provide a random base with exponents known only to the simulator. This idea gives rise to the RO-J-PAKE protocol below.

The RO-J-PAKE Protocol Description. A mathematical description of RO-J-PAKE is shown in Fig. 1. Next, we rephrase the protocol informally. In the description below, we will assume that the client and server always check if the received message is well-formed and if the validity of NIZK proof holds under appropriate label.

After initialization in which public parameters are fixed and a password different from zero is shared between the client and server, the protocol runs in two phases. In the first phase, each party generates one group element and corresponding NIZK proof and sends them – along with its ID – to the other party. In the second phase, upon receiving the first message, both parties compute a common base D as $H_0(f(A, B, X_1, X_2))$. Then, each party computes and sends to other party its commit message that consists of $\alpha := (DX_2)^{x_1 pw}$ and corresponding NIZK proof π_α under label l_A in case of client, and $\beta := (DX_1)^{x_2 pw}$ and π_β under label l_B in case of server. Upon receipt of the second message, each party derives a shared secret K, which should be an element of group $\mathbb{G}$, and then a bit-string sk, which will act as a session key.

The purpose of function f is to preserve the symmetry and keep the protocol within two message rounds by making sure that both parties sort values identically and compute the same D. In Section 2.4, we discuss the instantiation of the hash function H_0, while H_1 can be seen as a computational randomness extractor (see Section 3.2).

It is worth mentioning that RO-J-PAKE's design prevents the weird-but-benign case of *swapping instances* which happens in the original J-PAKE protocol if the values X_1 and X_2 (or X_3 and X_4 in case of server) are flipped. In that case, the NIZK proof π_β (or π_α resp.) from second message round would still be valid (since the base for the β and α values stay as intended), however, the derived keys would not be the same. A simple solution, proposed in [6], is to expand the NIZK proof labels and add to them all the received values. In RO-J-PAKE, the

---

[2]To be exact, we should also include pw into the formula for computing the base T_K and thus have $g^{(x_1 + x_3)pw}$, but this does not change our claim.

<div style="border:1px solid;">

Client A		Server B

Initialization

Public: $\mathbb{G}, g, q;\ \ H_0 : \{0,1\}^* \rightarrow \mathbb{G};\ \ H_1 : \{0,1\}^* \rightarrow \{0,1\}^k$

Secret: $pw \in \mathbb{Z}_q, pw \neq 0$

$$x_1 \leftarrow \mathbb{Z}_q \qquad\qquad x_2 \leftarrow \mathbb{Z}_q$$
$$X_1 := g^{x_1} \qquad\qquad X_2 := g^{x_2}$$
$$\pi_1 \leftarrow PK((X_1, g), x_1, A) \qquad\qquad \pi_2 \leftarrow PK((X_2, g), x_2, B)$$

$$\xrightarrow{\quad A, X_1, \pi_1 \quad}$$
$$\xleftarrow{\quad B, X_2, \pi_2 \quad}$$

abort if $X_2 = 1$ abort if $X_1 = 1$

check $VK((X_2, g), \pi_2, B)$ check $VK((X_1, g), \pi_1, A)$

$$D := H_0(f(A, B, X_1, X_2)) \qquad\qquad D := H_0(f(A, B, X_1, X_2))$$
$$\alpha := (DX_2)^{x_1 pw} \qquad\qquad \beta := (DX_1)^{x_2 pw}$$
$$\pi_\alpha \leftarrow PK((\alpha, DX_2), x_1 pw, l_A) \qquad\qquad \pi_\beta \leftarrow PK((\beta, DX_1), x_2 pw, l_B)$$

$$\xrightarrow{\quad \alpha, \pi_\alpha \quad}$$
$$\xleftarrow{\quad \beta, \pi_\beta \quad}$$

check $VK((\beta, DX_1), \pi_\beta, l_B)$ check $VK((\alpha, DX_2), \pi_\alpha, A, l_A)$

$$K := (\beta X_2^{-x_1 pw})^{x_1} \qquad\qquad K := (\alpha X_1^{-x_2 pw})^{x_2}$$
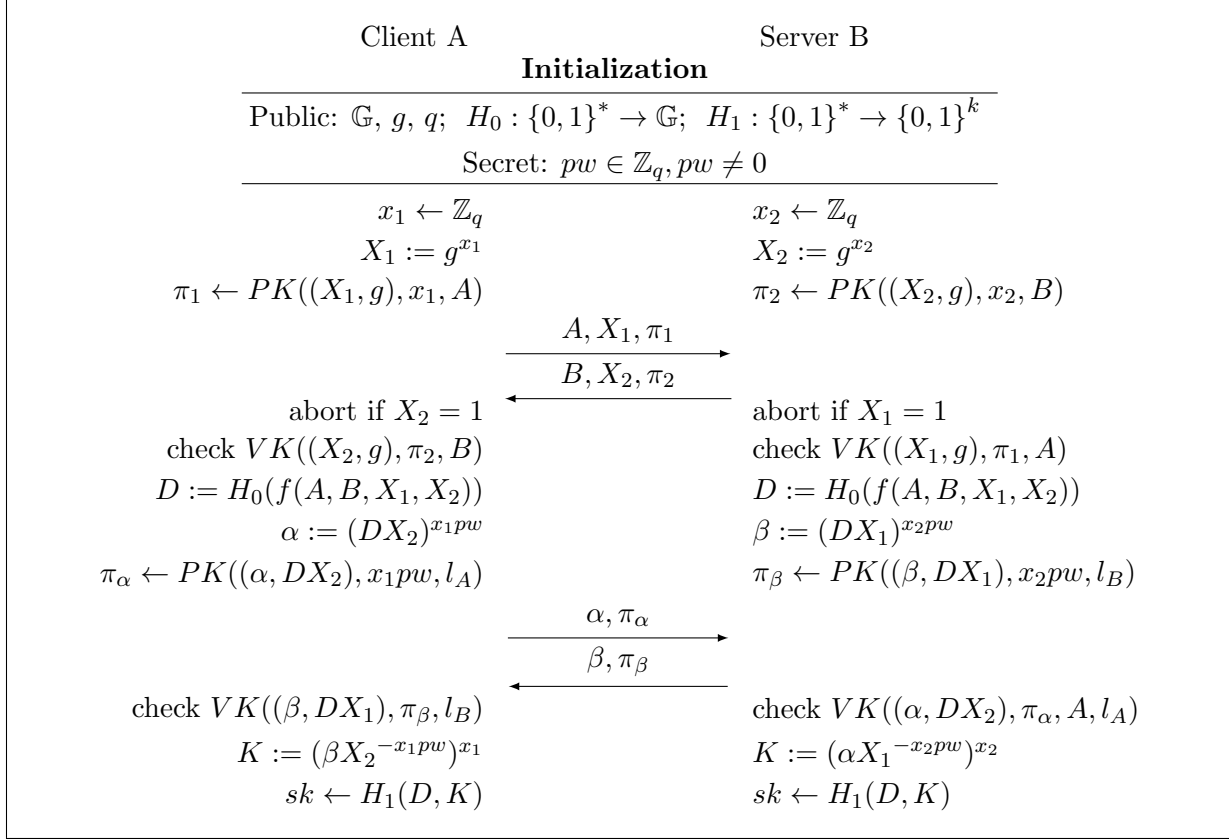$$sk \leftarrow H_1(D, K) \qquad\qquad sk \leftarrow H_1(D, K)$$

</div>

Figure 1: The RO-J-PAKE protocol. The value of labels are $l_A := (A, B, X_1, X_2)$ and $l_B := (B, A, X_2, X_1)$. PK generates NIZK proofs and VK verifies them.

swapping case does not occur even with the labels left out. However, we strongly advise using the labels in NIZK proofs to ensure that the messages from different rounds are bound together. This additionally makes the proof significantly tighter.

2.3   The CRS-J-PAKE Protocol

The observation that J-PAKE's $X_1 X_3$ value can be in a sense replaced by a random group element that neither party has control over can be exploited in another direction as well: We can simply add to the protocol's setup a randomly generated value $U \in \mathbb{G}$ that is fixed once and for all, and plays the role of $X_1 X_3$ in J-PAKE and D in RO-J-PAKE for all protocol executions. Hence, we can also consider the CRS-J-PAKE protocol, described fully below. Just like RO-J-PAKE, we eliminate two of the NIZK proofs by design. The name comes from the value U, which is a Common Reference String (CRS). In particular, it carries with it an underlying secret - i.e. the discrete log u of U to the base g - which must be unknown to all parties. In the security proof however, the simulator does get access to u, similarly to the way it knows the discrete logs of the outputs of hash values in the case of RO-J-PAKE (by programming the RO in this way).

Since we no longer need to hash into the underlying group, in contrast to RO-J-PAKE, CRS-J-PAKE has no efficiency issues with respect to a hash implementation. However, the need to generate and trust the hard-coded value U poses its own deployment issues (see Section 2.4).

The CRS-J-PAKE Protocol Description. CRS-J-PAKE is shown in Fig. 2. In comparison to RO-J-PAKE, the major difference is the adoption of the common reference string U, which will be securely chosen in the initialization phase and be hard-coded into the protocol implementation. As in RO-J-PAKE, swapping instance case does not occur by design.
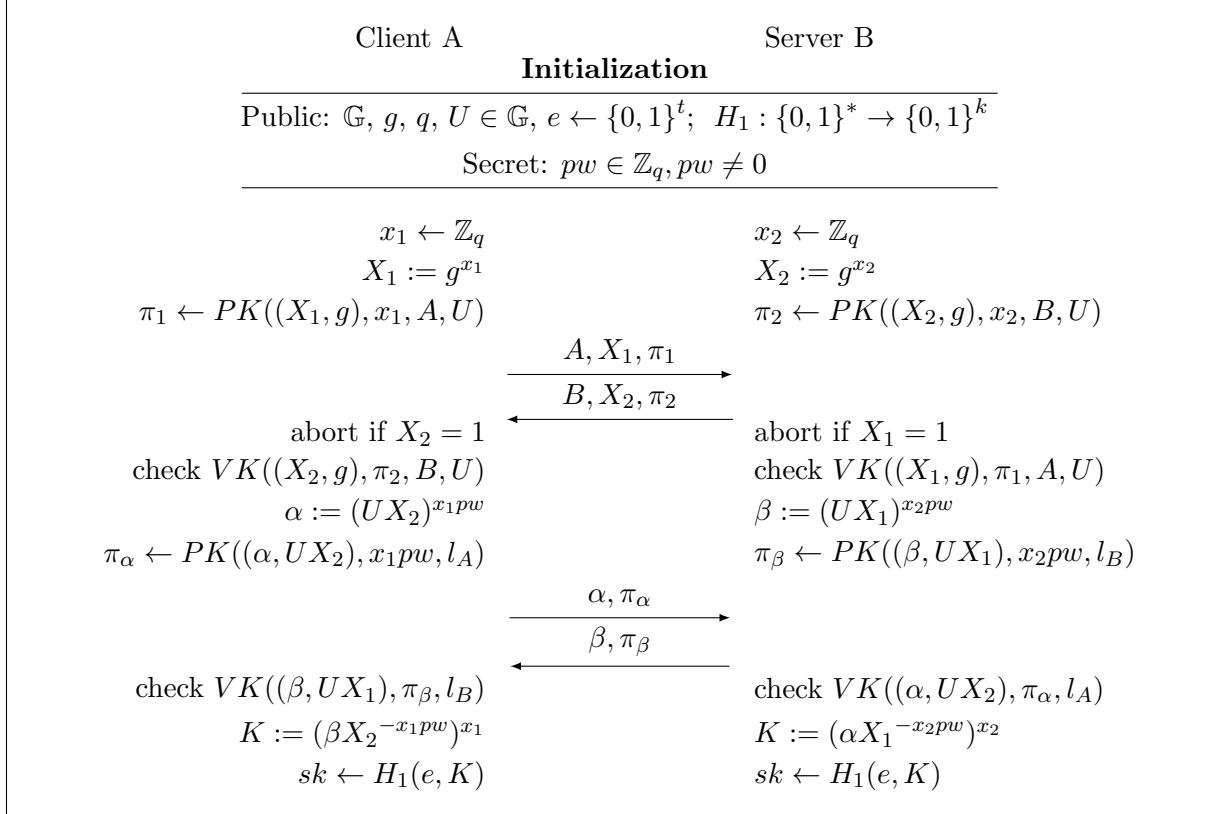
<div style="border:1px solid">

Client A   Server B

Initialization

Public: $\mathbb{G}, g, q, U \in \mathbb{G}, e \leftarrow \{0,1\}^t; \ H_1 : \{0,1\}^* \rightarrow \{0,1\}^k$

Secret: $pw \in \mathbb{Z}_q, pw \neq 0$

$$x_1 \leftarrow \mathbb{Z}_q \qquad\qquad x_2 \leftarrow \mathbb{Z}_q$$
$$X_1 := g^{x_1} \qquad\qquad X_2 := g^{x_2}$$
$$\pi_1 \leftarrow PK((X_1, g), x_1, A, U) \qquad \pi_2 \leftarrow PK((X_2, g), x_2, B, U)$$

$$\xrightarrow{\quad A, X_1, \pi_1 \quad}$$
$$\xleftarrow{\quad B, X_2, \pi_2 \quad}$$

$$\text{abort if } X_2 = 1 \qquad\qquad \text{abort if } X_1 = 1$$
$$\text{check } VK((X_2, g), \pi_2, B, U) \qquad \text{check } VK((X_1, g), \pi_1, A, U)$$
$$\alpha := (UX_2)^{x_1 pw} \qquad\qquad \beta := (UX_1)^{x_2 pw}$$
$$\pi_\alpha \leftarrow PK((\alpha, UX_2), x_1 pw, l_A) \qquad \pi_\beta \leftarrow PK((\beta, UX_1), x_2 pw, l_B)$$

$$\xrightarrow{\quad \alpha, \pi_\alpha \quad}$$
$$\xleftarrow{\quad \beta, \pi_\beta \quad}$$

$$\text{check } VK((\beta, UX_1), \pi_\beta, l_B) \qquad \text{check } VK((\alpha, UX_2), \pi_\alpha, l_A)$$
$$K := (\beta X_2^{-x_1 pw})^{x_1} \qquad\qquad K := (\alpha X_1^{-x_2 pw})^{x_2}$$
$$sk \leftarrow H_1(e, K) \qquad\qquad sk \leftarrow H_1(e, K)$$

</div>

Figure 2: The CRS-J-PAKE protocol. The client label $l_A := (A, B, X_1, X_2, U)$ and the server label $l_B := (B, A, X_2, X_1, U)$.

2.4   Practical Considerations

In theory, for J-PAKE and the two new variants, the modular exponentiations are the predominant factors in the computation. Hence, the computational cost is estimated based on counting the number of such modular exponentiations. Note that it takes one exponentiation to generate a Schnorr NIZK proof and two to verify it [31]. Referring to the protocol specifications in Fig. 1, 2, and 5, we summarize their complexities in Table 1.

In practice however, counting the modular exponentiations is insufficient, in particular for RO-J-PAKE. This is because the true speed depends highly on how H_0 - which lands into the protocol's underlying group - is computed. This is known to be less efficient than just hashing into a set of bitstrings and sometimes tricky to implement, especially in PAKEs. For instance, in protocols such as PAK [13], SPEKE [21] and Dragonfly [20, 26], a password is used as an input to a hash function, which means that the whole hashing procedure must be done in a constant time, otherwise side-channel attacks are possible (i.e. timing attack). This is not the

Table 1: The efficiency comparison between J-PAKE, RO-J-PAKE and CRS-J-PAKE.

Protocol	Complexity	
	Communication	Computation
J-PAKE	$12 \times \mathbb{G} + 6 \times \mathbb{Z}_q$	$28|q|$-bit exp
RO-J-PAKE	$8 \times \mathbb{G} + 4 \times \mathbb{Z}_q$	$20|q|$-bit exp + 2 H_0
CRS-J-PAKE	$8 \times \mathbb{G} + 4 \times \mathbb{Z}_q$	$20|q|$-bit exp

case with RO-J-PAKE, since all the inputs to the hash function are public values, but we still need to address the efficiency concern. Thus, we further discuss the computational complexity with respect to two different instantiations. It is important to recall that for the security proofs to be valid, $\mathbb{G}$ must be such that the DDH assumption is believed to hold, see [12] for examples.

- **SFF instantiation.** Here, we assume that $\mathbb{G}$ is deployed as the q-order subgroup of $GF(p)^*$, where $p = rq + 1$ and p and q are both prime. Thus, we have $|r| = |p| - |q|$. Standard techniques implement H_0 by first hashing into $GF(p)^*$, which is truly cheap, and then *exponentiating the result by r*, which depends on $|r|$. In particular Table 1, indicates that J-PAKE is more efficient than RO-J-PAKE if and only if $28|q| \leq 20|q| + 2|r|$, i.e. if and only if $4|q| \leq |r|$. In other words, J-PAKE is better than RO-J-PAKE provided that a single $|r|$-bit exponentiation costs more than 4 $|q|$-bit ones. Since Table 2 shows that in general, $|r|$-bit exponentiations cost *way more* than that, J-PAKE is definitely the better option when using SFFs[3]. Note that CRS-J-PAKE would be better than J-PAKE in this setting.

- **EC instantiation.** We carried out an experiment based on a Win7 64-bit operating system, with Intel(R) Core(TM) i7-5600U CPU@2.60GHz and 8.0GB RAM. In our test, we assumed the EC is over prime field $GF(p)$ with $|p| = 256$, and took $\mathbb{G}$ to be an EC group of prime order q with $|q| > 160$. H_0 was implemented using the recently discovered hashing algorithms of Brier et al. [14]. We found that an exponentiation takes on average 0.001383 seconds, while hashing a message into the EC group only takes 0.000086 seconds. (For reference, the source codes are listed in Appendix C.) This shows that hashing is about 16 times cheaper than exponentiating. Hence, using ECs, both RO-J-PAKE and CRS-J-PAKE are definitely more efficient than J-PAKE.

Further deployment notes for practitioners. On one hand, in favor of the new protocols, both are most-likely patent-free, like their big brother. Indeed, the structure of all three is essentially the same, having nothing really to do with that of EKE [11] or SPEKE [21]. For instance, none of the "J-PAKE"s perform any password-keyed encryption (like EKE) nor do they hash the password to get a commonly agreed-upon base (like SPEKE). The password is not even encrypted, as is done in many PAKEs that are standard-model-secure, e.g. [23, 22].

On the other hand, RO-J-PAKE and CRS-J-PAKE also have specific implementation issues to deal with for their security proofs to be of any use.

---

[3]Table 2 contains some NIST-recommended parameters, but even in theory the situation seems hopeless. Indeed, from [12] we see that for the DDH to reasonably hold in a SFF, we actually need $10|q| > |p| = |r|$, rendering the $4|q| \leq |r|$ requirement unachievable.

Table 2: Cost of an $|r|$-bit exponentiation compared to a $|q|$-bit one. E.g, for a 2048-bit modulus and 224-bit exponents, one $|r|$-bit exponentiation costs a bit over 8 $|q|$-bit ones. One sees the ratio getting much worse as the NIST-recommended [1] parameters grow.

$|p|$	$|q|$	$|r|$	$|r|/|q|$
1024	160	856	5.35
2048	224	1824	8.14
3072	256	2816	11

- **The random oracle model.** All three protocols' theoretical security relies on the random oracle model which is implicitly present in the security of the Schnorr NIZK proofs. However, RO-J-PAKE uses it arguably more than the other two, because of H_0. This does not point to any particular weakness, but care must always be taken when selecting the hash function in practice. It also introduces a additional degradation factor in the security proof.

- **The CRS.** It is important to understand that CRS-J-PAKE's security relies crucially on the CRS U being generated *randomly* and *such that $log_g(U)$ remains unknown to attackers*. This should be done in a trustworthy way [18]. For instance, a trusted authority can be asked to generate U by selecting u at random, setting $U = g^u$, and throwing u away, or even selecting a purely random string μ, and checking that μ encodes good U, without needing to "handle" u at all. This is an option for large institutions trying to deploy this protocol internally for employees. Another option would be for a predetermined set of users to jointly compute U, with the drawback that any additional user would have to trust the generated value.

3   Model

To prepare for the proofs, we outline the security model from [8] and present the complexity assumptions and cryptographic building blocks.

3.1   Model

Participants, Passwords and Initialization. Each principal U comes from either the *Clients* or *Servers* set, which are finite, disjoint, nonempty sets. We assume that each client $A \in Clients$ is in possession of a password pw_A, while each server $B \in Servers$ holds a vector of the passwords of all clients $pw_B = \langle pw_A \rangle_{A \in Clients}$. Before the execution of a protocol, an initialization phase occurs, in which public parameters are fixed and for each client a secret pw_A is drawn uniformly (and independently) at random from a finite set **Passwords** of size N and given to all servers.

Protocol Execution. The protocol P specifies how principals react to network messages. Since in reality each principal may run multiple executions of P with different partners, each principal

is allowed an unlimited number of *instances* executing P. We denote client instances by A^i and server instances by B^j. In this model, a bit b is flipped at the beginning of the game. To assess P's security, we assume that an adversary $\mathcal{A}$ has total network control, i.e. $\mathcal{A}$ provides inputs to instances, via the following *queries*:

- **Send**(U^i, M): $\mathcal{A}$ sends message M to instance U^i. As a response, U^i processes M according to P and outputs a reply. This query models active attacks.

- **Execute**(A^i, B^j): This triggers an honest run of P between A^i and B^j, and its transcript is given to $\mathcal{A}$. It covers passive eavesdropping on protocol flows.

- **Reveal**(U^i): $\mathcal{A}$ receives the current value of the session key sk_U^i. This captures session key leakage.

- **Test**(U^i): If $b = 1$, $\mathcal{A}$ gets sk_U^i. Otherwise, it receives a random string from the session key space. This query measures sk_U^i's semantic security.

- **Corrupt**(U): pw_U is given to $\mathcal{A}$. This models compromise of the long-term key.

Partnering. An instance U^i accepts if it holds a session key sk_U^i, a session ID sid_U^i and a partner ID pid_U^i. An instance U^i terminates if it will not send nor receive any more messages. Instances A^i and B^j are partnered if: (1) both accept; (2) $sid_A^i = sid_B^j \neq \perp$; (3) $pid_A^i = B$ and $pid_B^j = A$; (4) $sk_A^i = sk_B^j$; and (5) no other instance accepts with the same *sid*.

Freshness. Freshness captures the idea that the adversary should not trivially know the session key being tested. An instance U^i is said to be fresh with forward secrecy if: (1) it accepts; (2) no **Reveal** query was made to U^i nor to its partner U'^j; and (3) no **Corrupt**(U') query was made before U^i defined its session key sk_U^i, and a **Send**(U^i, M) query was made at some point, for any U'.

Advantage of the Adversary. We say that $\mathcal{A}$ wins and breaks the ake security of P, if upon making **Test** queries to fresh instances U^i that have terminated, $\mathcal{A}$ outputs a bit b', such that $b' = b$, where b is the bit selected at the beginning of the protocol. We denote the probability of this event by $\mathbf{Succ}_P^{ake}(\mathcal{A})$. The *ake*-advantage of $\mathcal{A}$ in breaking P is $\mathbf{Adv}_P^{ake}(\mathcal{A}) = 2\,\mathbf{Succ}_P^{ake}(\mathcal{A}) - 1$.

3.2 Cryptographic Building Blocks

We state the hardness assumptions upon which the security of our protocols rests, and introduce other useful building blocks.

Let $\mathcal{D}$ be a probabilistic algorithm trying to break a hardness assumption while running in time t and let $\varepsilon \in [0, 1]$. We say that the assumption holds over $\mathbb{G}$ if there does not exist a (t, ε)-solver for polynomial t and non-negligible ε. For any x, y and z from $\mathbb{Z}_q$, set $DH_g(g^x, g^y) := g^{xy}$, $SDH_g(g^x) := g^{x^2}$ and $TGDH_g(g^x, g^y, g^z) := g^{xyz}$. Let $\mathcal{C}$ be a challenger.

Decision Diffie-Hellman (DDH). We say that $\mathcal{D}$ is a (t, ε)-DDH solver if $\mathbf{Adv}_{g,\mathbb{G}}^{ddh}(\mathcal{D}) := \mathbf{Succ}_{g,\mathbb{G}}^{ddh}(\mathcal{D}) - \frac{1}{2} \geq \varepsilon$, where $\mathbf{Succ}_{g,\mathbb{G}}^{ddh}(\mathcal{D}) := \Pr[b' = b]$ in the following game.

$\mathcal{C}$ flips a bit b, and chooses uniformly at random values x, y, and z in $\mathbb{Z}_q$. Then, $X := g^x$ and $Y := g^y$ are computed and, Z is set as follows: $Z := g^z$ if b is equal to 0 and $Z := DH_g(X, Y)$ otherwise. Now, $\mathcal{D}$ gets as input (g, X, Y, Z) and tries to distinguish whether Z is the real

Diffie-Hellman value $DH_g(X, Y)$ or a random group element of $\mathbb{G}$. At the end of the game, $\mathcal{D}$ outputs a bit b'.

Decision Square Diffie-Hellman (DSDH). We say $\mathcal{D}$ is a (t, ε)-DSDH solver if $\mathbf{Adv}_{g,\mathbb{G}}^{dsdh}(\mathcal{D}) :=$ $\mathbf{Succ}_{g,\mathbb{G}}^{dsdh}(\mathcal{D}) - \frac{1}{2} \geq \varepsilon$, where $\mathbf{Succ}_{g,\mathbb{G}}^{dsdh}(\mathcal{D}) := \Pr[b' = b]$ in the following game.
    First x and y are chosen uniformly at random from $\mathbb{Z}_q$ and a bit b is flipped by $\mathcal{C}$. Let $X := g^x$. If the bit b that $\mathcal{C}$ holds is equal to 0, then $Y := g^y$. Otherwise, set $Y := SDH_g(X)$. Now, $\mathcal{D}$ gets as input (g, X, Y) and tries to distinguish whether Y is a square Diffie-Hellman value or a random group element of $\mathbb{G}$. At the end of the game, $\mathcal{D}$ outputs a bit b'.

Decision Triple Group Diffie-Hellman (DTGDH). We say $\mathcal{D}$ is a (t, ε)-DTGDH solver if $\mathbf{Adv}_{g,\mathbb{G}}^{dtgdh}(\mathcal{D}) := \mathbf{Succ}_{g,\mathbb{G}}^{dtgdh}(\mathcal{D}) - \frac{1}{2} \geq \varepsilon$, where $\mathbf{Succ}_{g,\mathbb{G}}^{dtgdh}(\mathcal{D}) := \Pr[b' = b]$ in the following game.
    $\mathcal{C}$ chooses x, y, z, and w uniformly at random in $\mathbb{Z}_q$ and flips a bit b. $\mathcal{C}$ computes $X := g^x$, $Y := g^y$, and $Z := g^z$. The value W is set to g^w if $b = 0$, or $W := TGDH_g(X, Y, Z)$ otherwise. $\mathcal{D}$ gets $(g, X, Y, Z, DH_g(X, Y), DH_g(X, Z), DH_g(Y, Z), W)$, and tries to tell whether W is a Triple Diffie-Hellman value or a random group element. At the end of the game, $\mathcal{D}$ outputs bit a b'.

Random Oracle. In the random oracle model [10], hash functions are modeled as public, random functions - with co-domain $\{0, 1\}^k$ or some particular group - that the adversary has query access to. Answers to new input are selected randomly, while answers to previous inputs are repeated, see Fig. 3.

Common Reference String (CRS). In the CRS model, a public, trusted value – called the CRS – is selected at setup time, and given to all participants and the adversary. To CRS may be associated an underlying trapdoor, which the simulator gets access to during the security proof.

Simulation-Sound Extractable NIZK Proofs (SE-NIZK). We keep the discussion here informal; for more details on SE-NIZK, we refer to [17] and [6].
    Let $\mathcal{R}$ be an efficiently computable relation with a binary output and two inputs (x, w), where x and w are called the statement and the witness, respectively. Let $\mathcal{L}$ be the NP-language that consist of statements with respect to $\mathcal{R}$: $\mathcal{L} = \{x \mid \exists w, \mathcal{R}(x, w) = 1\}$. A NIZK proof system $(Setup, PK, VK)$ for $\mathcal{R}$ is a two-party protocol, where on input w a prover is able to prove to a verifier that some statement x is the member of $\mathcal{L}$ without revealing w. In practice, the prover produces a proof $\pi \leftarrow PK(x, w, l)$ for some label l. Anyone holding x, π, and l can check the proof by running algorithm $VK(x, \pi, l)$, which outputs 1 if the proof is valid, and 0 otherwise.
    If *(unbounded) zero-knowledge* (UZK) and *simulation-sound-extractability* (SE) both hold, we say that $(Setup, PK, VK)$ is SE-NIZK. Informally, UZK ensures that simulated proofs are indistinguishable from real one, while SE guarantees that there exists an *Ext* algorithm that can extract a witness from any adversary-generated proof, even if the adversary can see simulated proofs. These properties are typically enabled in NIZK proof systems by generating a trapdoor for some additional CRS at setup time. However, this is not the case for Schnorr proofs [31], which are used to instantiate SE-NIZK in J-PAKE. Fortunately, it was shown in [6] that under certain conditions Schnorr proofs satisfy both properties: ZK stems from the programmability of the RO, while for SE the adversary has to be assumed *algebraic*, and all the bases used in

protocol must be hard-linear.

Computational Randomness Extractor. In the original J-PAKE paper [19], the hash function used for key derivation is implicitly modeled as a random oracle. However, it was shown in [6] that a computational randomness extractor for random group elements [25] is sufficient. Such a randomness extractor is a function $ext_R : \{0,1\}^t \times \mathbb{G} \rightarrow \{0,1\}^k$, for some $t \geq 0$. The extractor is said to be secure if a polynomial-time adversary $\mathcal{A}$'s advantage in distinguishing $ext_R(r, e)$ from a random bitstring in $\{0,1\}^k$ given r, and where (r, e) is randomly sampled from $\{0,1\}^t \times \mathbb{G}$, is negligible. For more details, see [6].

4   Security Analysis

In this section we present the security proofs for RO-J-PAKE and CRS-J-PAKE. Due to their similarity with the J-PAKE protocol, we are able to structure the proofs in the vein of [6] with the goal to simplify proofreading. The proofs that are demonstrated here are slightly simpler than the original J-PAKE proof. This is true even in case the labels l_A and l_B only contain the identity of the originator of the NIZK proofs π_α and π_β (see Appendix B.1), as in the original J-PAKE.

Throughout our analysis, we will assume that the NIZK proofs used in are SE-NIZK. This is crucial, since it will allow the simulator to tell apart correct and incorrect password guesses and simulate all queries made by the adversary. As in J-PAKE, we keep Schnorr proofs of knowledge as the instantiation of SE-NIZK in our protocols. In [6], they are shown to be SE-NIZK in the algebraic adversary model with random oracles under one additional condition: the hard-linearity property of bases used in proof must be exhibited. Since the security of our protocols rests on the the same hardness assumptions as those in [6], the hard-linearity property of bases is preserved. Additionally, for the proofs to go through, it is as well crucial that the discrete logs of D in RO-J-PAKE (from the RO) and of U in CRS-J-PAKE (the CRS) are known to the simulator.

4.1   Proof of Security for RO-J-PAKE

To exhibit the security of RO-J-PAKE, we will bound the adversarial advantage in attacking the ake security of the studied protocols by using sequence-of-games approach. Starting from the original attack game $\mathbf{G}_0$ – which is played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ – we will make a small change to a corresponding protocol P_0 and thus define the next game. Our goal is to prove that $\mathcal{A}$'s advantage is proportional to that of the "dummy" online guesser by showing that $\mathcal{A}$ has negligible advantage to distinguish between two successive games with the exception of game $\mathbf{G}_4$, where guessing-the-right-password event occurs with non-negligible probability.[4]

Going further, the challenger $\mathcal{C}$ takes the role of a simulator that executes the protocol for $\mathcal{A}$. The protocol execution begins by an initialization phase(see Fig. 4). Then, the simulator gives to $\mathcal{A}$ all public values generated in the initialization phase. Upon receiving an oracle query from $\mathcal{A}$, $\mathcal{C}$ will respond by executing the appropriate algorithm as in Fig. 3. All state information generated during the execution of protocol will be recorded by the simulator.

---

[4]This is where the cost is paid for using a small entropy secret for authentication instead of cryptographically strong authenticator.

Theorem 1. *Consider **RO-J-PAKE** as specified in Fig. 1, with a password set of size N. Let $\mathcal{A}$ be an adversary that runs in time at most t, and makes at most n_{se}, n_{ex}, n_{re}, n_{te}, n_{h0} queries of type **Send**, **Execute**, **Reveal**, **Test** and RO queries to H_0. It holds that*

$$\boldsymbol{Adv}_{\mathtt{ro-j-pake}}^{ake}(\mathcal{A}) \leq \frac{n_{se}}{N} \;+\; O\bigg(\frac{(n_{se} + n_{ex} + n_{ho})^2}{q} \;+\; \frac{n_{h0}^2}{q} \;+\; \boldsymbol{Adv}_{g,\mathbb{G}}^{dsdh}(t')$$
$$+ (n_{ex} + n_{se}^2)\boldsymbol{Adv}_{g,\mathbb{G}}^{dtgdh}(t') \;+\; 2n_{h0}n_{se}\boldsymbol{Adv}_{g,\mathbb{G}}^{ddh}(t')$$
$$+ (n_{re} + n_{te})\boldsymbol{Adv}_{ext_R}^{comp}(t') \;+\; \boldsymbol{Adv}_{NIZK}^{uzk}(t') \;+\; \boldsymbol{Adv}_{NIZK}^{ext}(t')\bigg),$$

and where $t' = O(t + (n_{se} + n_{ex} + n_{ho})t_{exp})$ with t_{exp} being the time required for an exponentiation in $\mathbb{G}$.

Proof. From now on, the values that are received by an honest party and possibly coming from $\mathcal{A}$ will be denoted as X_1', α', etc. We say that instance is *matching* if $X_1 = X_1'$ and $X_2 = X_2'$. In that case, the client's hash output D_A will be equal to the server's D_B. Also, we say that instances are *fully matching*, if both message rounds are honestly forwarded by $\mathcal{A}$.

Game G_0 : (Original protocol) This game is faithful to Fig. 1.

Game G_1 : (Simulation and extraction) As defined in Sect. 3, we simulate **Send**, **Execute**, **Reveal**, **Corrupt**, and **Test** queries that $\mathcal{A}$ may make, with the difference now that for **Send** queries, the simulator runs an extractor Ext, which takes as input a NIZK proof that is produced by $\mathcal{A}$, and outputs a corresponding witness. If the extraction fails, so does $\mathcal{A}$. Also, all hash queries to H_0 are answered by maintaining a list $\mathcal{L}_{h0}$ (see Fig. 3).

From now on, we assume that an instance receiving a non-valid NIZK proof aborts. More importantly, the simulator – by running the extractor Ext – can obtain discrete logs x_1', x_2', $x_1'pw'$, and $x_2'pw'$ (and thus pw') from corresponding NIZK proofs that are generated by $\mathcal{A}$. Note that we assume that the simulator knows the discrete logarithms of the outputs of H_0 queries.

$$\mathbf{Adv}_{\mathtt{ro-j-pake}}^{ake}(\mathcal{A}) = \mathbf{Adv}_{\mathrm{P}_1}^{ake}(\mathcal{A}) + O\left(\mathbf{Adv}_{NIZK}^{uzk}(t') + \mathbf{Adv}_{NIZK}^{ext}(t')\right) \;. \tag{1}$$

H_0: For each hash query $H_0(w)$, if the same query was previously asked, the simulator retrieves the record (w, D, d) from the list $\mathcal{L}_{h0}$ and answers with D. Otherwise, the answer D is chosen according to the following rule:

⋆ **Rule $H_0^{(1)}$**
     Choose $d \leftarrow \mathbb{Z}_q$. Compute $D := g^d$ and write the record (w, D, d) to $\mathcal{L}_{h0}$.

Figure 3: Simulation of the hash function H_0

Game G_2 : (Force uniqueness and avoid collisions) In this game, collisions on the partial transcript $((A, X_1, \pi_1), (B, X_2, \pi_2))$ and the H_0 random oracle are avoided.

More precisely, if a value X_1 or X_2 is repeated in the protocol execution or has already appeared in the random oracle query made by $\mathcal{A}$, the protocol halts and $\mathcal{A}$ fails. The same happens if the outputs of distinct H_0 random oracle queries coincide. Both events are bounded with the birthday paradox:

$$\mathbf{Adv}_{P_1}^{ake}(\mathcal{A}) = \mathbf{Adv}_{P_2}^{ake}(\mathcal{A}) + O\left(\frac{(n_{se} + n_{ex} + n_{h0})^2}{q}\right) + O\left(\frac{n_{h0}^2}{q}\right) \ . \tag{2}$$

Game $\mathbf{G}_3$: (Allow instance linking) Same as $\mathbf{G}_2$.

As we can see in Fig. 1, the values A, B, X_1 and X_2 are all included in the labels l_A and l_B. This renders the game $\mathbf{G}_3$ from the J-PAKE proof unnecessary. Moreover, we show in Appendix B.1 that the security of RO-J-PAKE still holds even if labels that contain only the identity are used.

$$\mathbf{Adv}_{P_2}^{ake}(\mathcal{A}) = \mathbf{Adv}_{P_3}^{ake}(\mathcal{A}) \ . \tag{3}$$

Game $\mathbf{G}_4$: (Check password guesses) If before a **Corrupt** query, $\mathcal{A}$ makes a **Send** query to a non-matching instance containing α' or β' that corresponds to a correct password guess, the protocol halts and $\mathcal{A}$ succeeds.

The crucial observation here is that the simulator can check whether the password guess is correct or not. This is so, since the simulator can obtain discrete logs of X_1', X_2', α' and β' by running *Ext* on the corresponding NIZK proofs. The extraction does not work for the value coming from a reduction, which we will call a *simulated value*, otherwise the simulator could break the hardness assumption trivially. To determine if the password guess is correct, the simulator can proceed as follows: 1) if both X_1' and α' (or X_2' and β' in the case of server impersonation) come from $\mathcal{A}$, the simulator extracts two discrete logs from the corresponding NIZK proofs (e.g. x_1' from π_1 and $x_1'pw'$ from π_α), divides them and checks whether the result is equal to pw_A; or 2) if one of the values that instance receives is a simulated value (X_1' or α' and X_2' or β'), the simulator extracts one discrete log of the value coming from $\mathcal{A}$ and combines it with the correct password pw_A to perform a check against the simulated value.

$$\mathbf{Adv}_{P_3}^{ake}(\mathcal{A}) \leq \mathbf{Adv}_{P_4}^{ake}(\mathcal{A}) \ . \tag{4}$$

Game $\mathbf{G}_5$: (Randomize session keys for wrong password guesses) In case of an false password guess to a non-matching instance, K is set randomly.

The proof is split into two parts. In the first, we set K randomly only in the non-matching client instances in case of a wrong guess – we will call those *target* client instances. We construct an algorithm $\mathcal{D}$ that given a tuple $\langle X, Y \rangle$, where $X \leftarrow g^x$ and $Y \in \mathbb{G}$, attempts to break the DSDH assumption by running $\mathcal{A}$ as a subroutine. The algorithm $\mathcal{D}$ simulates the protocol for $\mathcal{A}$ by setting K randomly for all target client instances, and computing K normally for all other client instances as follows.[5]

---

[5]Due to the labels l_A and l_B, the random self-reducibility of the DSDH assumption can be used. This affects the tightness of the proof: we do not need to use **a** hybrid argument, so the factor n_{se} in front of $\mathbf{Adv}_{g,\mathbb{G}}^{dsdh}(\mathcal{D})$ in the Theorem 1 does not appear.

For a given DSDH instance $\langle X, Y \rangle$, any client instance A^i chooses $a, b \leftarrow \mathbb{Z}_q^*$ and sets $X_1 = X^a g^b$. When A^i receives a **Send**$(A^i, (B, X_2', \pi_2'))$ query, the simulator extracts x_2' from π_2', computes D_A and sets $\alpha = X^{a(d'+x_2')pw_A} g^{b(d'+x_2')pw_A}$. After receiving **Send**$(A^i, (\beta', \pi_\beta'))$ and extracting a witness from π_β', the client instance A^i computes $K = X^{ad'x_2'pw'} g^{bd'x_2'pw'}$ $Y^{a^2 x_2'(pw'-pw_A)} X^{2abx_2'(pw'-pw_A)} g^{b^2 x_2'(pw'-pw_A)}$. Note that the value pw' is obtained by dividing extracted witnesses from the NIZK proofs $pw' = Ext(\pi_\beta')/Ext(\pi_2')$, while d' comes from the list $\mathcal{L}_{h0}$ (see Fig. 3). In case of matching instances or a correct password guess $K = X^{d_A pw_A x_2'} g^{bd'x_2'pw'}$, since pw' is equal to pw_A. If Y is a real DSDH challenge, then the value K will be computed as in $\mathbf{G}_4$. On the other hand, if Y is random and an incorrect password guess is made, then K will be random, since $x_2' \neq 0$ (checked in the protocol, see Fig. 1) and $a(pw'-pw_A) \neq 0$.

Now we show that the simulation is sound for any instance of B that generates X_2 and receives possibly simulated X_1' or α'. If any of the two received values is not from A^i, the simulator can extract a witness from the NIZK proof and check whether the password is correct or not. Moreover, if both received values are from A^i, the instances are matching, otherwise π_α would not be valid. Thus, there is no need to check the password in this case. Since the reduction for the second part of proof in case of relevant server instances is analogous, we get the following bound:

$$\mathbf{Adv}_{P_4}^{ake}(\mathcal{A}) = \mathbf{Adv}_{P_5}^{ake}(\mathcal{A}) + \mathbf{Adv}_{g,\mathbb{G}}^{dsdh}(t') \ . \tag{5}$$

Game $\mathbf{G}_6$: (Randomize session keys for paired instances) In case of a matching instance, set K randomly (and matching instances get the same K).

We use DTGDH (see Sec. 3.2) in a hybrid argument. We build an algorithm $\mathcal{D}$ that randomly chooses indexes $i, j \leftarrow \{1, 2, \ldots, n_{se}\}$ and simulates the protocol by computing K randomly for all (lexicographically) previous instances $(A^{i'}, B^{j'})$ that are fully matching, and setting K normally for all later $(A^{i'}, B^{j'})$.

For (A^i, B^j), A^i sets $X_1 = X$, B^j sets $X_2 = Y$, while the value Z is embedded as the output of $H_0(A^i, B^j, X, Y)$. If A^i and B^j match, A^i sets $\alpha = (DH_g(X, Z)DH_g(X, Y))^{pw_A}$ and B^j sets $\beta = (DH_g(Y, Z)DH_g(X, Y))^{pw_A}$. If they fully match, the shared secret is computed as $K = W^{pw_A}$. If $W = g^{xyz}$ then this simulates computing K normally. If W is random, then K is random since $pw_A \neq 0$.

We now need to check if the simulation is sound for other possible queries to A^i and B^j. If A^i (resp. B^j) receives non-simulated values X_2' and β' (resp. X_1' and α'), it can extract witnesses from the NIZK proofs, check the password guess, and respond accordingly. If A^i (resp. B^j) receives an X_2' (resp. X_1') value from $\mathcal{A}$ and a simulated β (resp. α), π_β (resp. π_α) would not be valid due to the labels l_B (resp. l_A). Conversely, if after the first message flow the instances are matching ($X_1 = X_1'$ and $X_2 = X_2'$), but α' or β' are from $\mathcal{A}$, the password guess will be wrong. In case a password guess is correct and the simulation continues, a **Corrupt** query has been made due to $\mathbf{G}_4$. Since the simulator can extract x_1' or x_2' in case of impersonation, K can be computed as $\mathcal{A}$ expects it to be. Therefore,

$$\mathbf{Adv}_{P_5}^{ake}(\mathcal{A}) = \mathbf{Adv}_{P_6}^{ake}(\mathcal{A}) + (n_{ex} + n_{se}^2)\mathbf{Adv}_{g,\mathbb{G}}^{dtgdh}(t') \ . \tag{6}$$

Game $\mathbf{G}_7$: (Randomize α and β) If there was no **Corrupt** query, set α and β randomly in all instances. In the case of a correct password guess to a non-matching instance (after **Corrupt**

query), compute K as the other party would.

Again the proof is split into two parts. Firstly, using a reduction from DDH, the α values are set randomly. We construct an algorithm $\mathcal{D}$ that randomly chooses $i \leftarrow \{1, 2, \ldots, n_{se}\}$ and $j \leftarrow \{1, 2, \ldots, n_{ho}\}$, and simulates the protocol for $\mathcal{A}$ by setting an exponent $x_1 pw_A$ in α to be random for all client instances prior to A^i and computing α normally for all client instances after A^i as follows.[6]

A^i sets $X_1 = X$, and the challenge Y is embedded as the output of the jth $H_0(A, B, X_1, X_2')$ query. After receiving $\textbf{Send}(A^i, (B, X_2', \pi_2'))$, the simulator extracts x_2' to compute $\alpha = Z^{pw_A} X^{x_2' pw_A}$. Clearly, if Z is random, so is α, and if $Z = DH_g(X, Y)$, α is computed as in $\textbf{G}_6$. If $\mathcal{A}$ succeeds (by guessing b or the correct password), $\mathcal{D}$'s guess to the DDH challenger is $b_1' = 1$, and 0 otherwise.

Note that upon receiving β which corresponds to a correct password guess (either from $\mathcal{A}$ or from B), the simulator can make other instances of A compute the key as $\mathcal{A}$ or B would, even if α is random. This is possible since it can extract x_2', $x_2' pw'$ from the proofs, check the password guess, and run A accordingly.

We now show the simulation is sound for any instance of B that generates X_2 and receives a possibly simulated X_1' or α' from A^i: 1) if both values are from A^i, set K randomly (due to $\textbf{G}_6$); 2) if either value is not from A^i, the simulator can extract witnesses from the proof, checks if the password is correct (and satisfies $\textbf{G}_4$) or not (and sets K randomly due to $\textbf{G}_5$); 3) if some α' corresponding to a correct password guess submitted to an instance of B is not from A^i and the execution continues (a $\textbf{Corrupt}$ query has been made), the discrete log of X_1 is known and the simulator can compute the shared secret K_B as $\mathcal{A}$ would.

Since the reduction for the second part of proof in case of relevant server instances is analogous, we get the following bound:

$$\textbf{Adv}_{P_6}^{ake}(\mathcal{A}) = \textbf{Adv}_{P_7}^{ake}(\mathcal{A}) + 2n_{h0} n_{se} \textbf{Adv}_{g,\mathbb{G}}^{ddh}(t') \ . \tag{7}$$

Game $\textbf{G}_8$: (Randomize sk) Set sk randomly in all instances in which K is set randomly (the matching instances get the same sk).

Remember that sk is computed as $H_1(D, K)$ and that D is the output of a random oracle. The games are computationally indistinguishable, since K is random and H_1 is a computational randomness extractor.

$$\textbf{Adv}_{P_7}^{ake}(\mathcal{A}) = \textbf{Adv}_{P_8}^{ake}(\mathcal{A}) + (n_{re} + n_{te}) \textbf{Adv}_{ext_R}^{comp}(t') \ . \tag{8}$$

This concludes the proof.

$\square$

4.2   Proof of Security for CRS-J-PAKE

Due to its very high similarity with the proof from Section 4.1, we will only show a sketch of the security proof for CRS-J-PAKE. The main idea behind the proof is that instead of knowing the discrete logs of H_0's output, the simulator knows the discrete log of parameter U.

---

[6]In addition to factor n_{se}, which appears in the proof of original J-PAKE [6], there is a security degradation of factor n_{ho}, since in this reduction the simulator also needs to guess the 'right" random oracle query.

Theorem 2. *Consider* ***CRS-J-PAKE*** *(see Fig. 2) with a password set of size N and fixed public value U. Let $\mathcal{A}$ be an adversary that runs in time at most t, and makes at most n_{se}, n_{ex}, n_{re}, n_{te}, n_{h0} queries of type* ***Send***, ***Execute***, ***Reveal***, *and* ***Test***. *It holds that*

$$
\begin{aligned}
\boldsymbol{Adv}_{\texttt{crs-j-pake}}^{ake}(\mathcal{A}) \leq \frac{n_{se}}{N} \; &+ \; O\Bigg(\frac{(n_{se} + n_{ex})^2}{q} \; + \; (n_{ex} + n_{se}^2)\boldsymbol{Adv}_{g,\mathbb{G}}^{dtgdh}(t') \\
&+ \; \boldsymbol{Adv}_{g,\mathbb{G}}^{dsdh}(t') \; + \; 2n_{se}\boldsymbol{Adv}_{g,\mathbb{G}}^{ddh}(t') \; + \; (n_{re} + n_{te})\boldsymbol{Adv}_{ext_R}^{comp}(t') \\
&+ \; \boldsymbol{Adv}_{NIZK}^{uzk}(t') \; + \; \boldsymbol{Adv}_{NIZK}^{ext}(t') \Bigg),
\end{aligned}
$$

where $t' = O(t + (n_{se} + n_{ex} + n_{ho})t_{exp})$ with t_{exp} being the time required for an exponentiation in $\mathbb{G}$.

Proof Summary. At first, the value of the public parameter U is fixed in the initialization phase and its discrete log u is discarded. A first distinction with the RO-J-PAKE proof occurs in game $\mathbf{G}_2$, where we do not need to avoid collisions on the function H_0. Then, in game $\mathbf{G}_3$, the initialization phase is changed as shown in Fig. 4. This means that the discrete log u of the public parameter U is now saved and can be used by the simulator during the protocol execution. Game $\mathbf{G}_4$ stays the same as in the RO-J-PAKE proof.

    After the initial four games, we come to the "reduction" games $\mathbf{G}_5$, $\mathbf{G}_6$, and $\mathbf{G}_7$, where we use similar reductions from DSDH, DTGDH and DDH respectively (as in RO-J-PAKE proof), but with a minor differences. In all three reductions, the value u is used during the simulation to compute α and K values, instead of using d_A as in RO-J-PAKE. Also, in $\mathbf{G}_6$, the Z value from a DTGDH challenge tuple is inserted in place of U instead of being inserted as the output of H_0. Similarly, the value Y that comes from the DDH challenge tuple in $\mathbf{G}_7$ is inserted in place of U instead of in the output of H_0. This last change removes the security degradation factor n_{ho} which appears in RO-J-PAKE's bound. Finally, the public random string e, which is generated during the initialization phase, allows us to set sk randomly in all instances in which K is set randomly and to argue the computational indistinguishability between games $\mathbf{G}_7$ and $\mathbf{G}_8$.

Proof Sketch. We will keep the meaning and fully-matching instances as in the proof of RO-J-PAKE. The value of the parameter U is fixed in the initialization phase and its discrete log u is discarded.

Game $\mathbf{G}_0$: (Original protocol) This game is faithful to Fig. 2.

Game $\mathbf{G}_1$: (Simulation and extraction) Same as in Sec. 4.1.

Game $\mathbf{G}_2$: (Force uniqueness and avoid collisions) In this game, collisions on the partial transcript $((A, X_1, \pi_1),(B, X_2, \pi_2))$ are avoided.

We do not need to avoid collisions on the hash function H_0 as in RO-J-PAKE and therefore

$$
\mathbf{Adv}_{\mathrm{P}_1}^{ake}(\mathcal{A}) = \mathbf{Adv}_{\mathrm{P}_2}^{ake}(\mathcal{A}) + O\left(\frac{(n_{se} + n_{ex})^2}{q} \right) \quad . \tag{9}
$$

Game $\mathbf{G}_3$: (Keep the discrete log of public parameter) During the initialization phase, the discrete log u of the public parameter U is saved for future use.

At this point, the change is made in the initialization procedure as in Fig. 4. During the protocol execution, the simulator can retrieve the record (U, u) from $\mathcal{L}_U$ if necessary.

$Initialization(1^k)$: Let $\mathbb{G}$ be a finite multiplicative group of prime order q, and g be a generator of $\mathbb{G}$. *Clients* and *Servers* sets are well defined. The initialization procedure is performed as follows:

⋆ **Rule** $Initialization^{(3)}$

    Choose $u \leftarrow \mathbb{Z}_q^*$. Compute $U := g^u$ and write the record (U, u) to $\mathcal{L}_U$.

    Choose $e \leftarrow \{0, 1\}^t$.

    For each $A \in Clients$: $pw_A \leftarrow$ **Passwords**.

    For each $B \in Servers$: $pw_{A,B} = pw_A$.

    Return $\mathbb{G}, g, q, e, A, B, U$.

Figure 4: Initialization procedure

$$\mathbf{Adv}_{\mathrm{P}_2}^{ake}(\mathcal{A}) = \mathbf{Adv}_{\mathrm{P}_3}^{ake}(\mathcal{A}) \ . \tag{10}$$

Game $\mathbf{G}_4$: (**Check password guesses**) If before a **Corrupt** query, $\mathcal{A}$ makes a **Send** query to a non-matching instance containing α' or β' that corresponds to a correct password guess, the protocol halts and $\mathcal{A}$ succeeds.

As in $\mathbf{G}_4$ from the RO-J-PAKE proof (see Sec. 4.1), by using the extraction property of SE-NIZK, the simulator can check whether the password guess is correct or not.

Game $\mathbf{G}_5$: (**Randomize session keys for wrong password guesses**) In case of an incorrect password guess to a non-matching instance, K is set randomly.

In this game, as in RO-J-PAKE proof, we use the random self-reducibility of DSDH to show that the adversary needs to solve DSDH in order to distinguish between the two games. There is only a minor difference with the RO-J-PAKE proof of $\mathbf{G}_5$. Namely, during the simulation, the value u is used instead of d_A when computing the α and K values.

As before, we split the proof in two parts. In the first part of the proof, we set K randomly only in the non-matching client instances in case of a wrong password guess – we will call those *target* client instances. We construct an algorithm $\mathcal{D}$ that given a tuple $\langle X, Y \rangle$, where $X \leftarrow g^x$ and $Y \in \mathbb{G}$, attempts to break the DSDH assumption (i.e. determine whether Y is random or $Y = g^{x^2}$) by running the adversary $\mathcal{A}$ as a subroutine. The algorithm $\mathcal{D}$ simulates the protocol for $\mathcal{A}$ by setting K randomly for all target client instances, and computing K normally for all other client instances as follows.

For a given DSDH instance $\langle X, Y \rangle$, any client instance A chooses $a, b \leftarrow \mathbb{Z}_q$ and sets $X_1 = X^a g^b$. When A^i receives a **Send**$(A^i, (B, X_2', \pi_2'))$ query, the simulator extracts x_2' from π_2', and sets $\alpha = X^{a(u+x_2')pw_A} g^{b(u+x_2')pw_A}$. After receiving a **Send**$(A^i, (\beta', \pi_\beta'))$ and extracting a witness from π_β', the client instance A^i computes $K = X^{aux_2'pw'} g^{bux_2'pw'} Y^{a^2 x_2'(pw'-pw_A)} X^{2abx_2'(pw'-pw_A)} g^{b^2 x_2'(pw'-pw_A)}$. Note that the value pw' is obtained by dividing the extracted witnesses from the NIZK proofs $pw' = Ext(\pi_\beta')/Ext(\pi_2')$, while u comes from the record $\mathcal{L}_U$ (see Fig. 4). In

case of matching instances or a correct password guess $K = X^{upw_A x_2'} g^{bux_2' pw'}$, since pw' is equal to pw_A. If Y is a real DSDH challenge, then the value K will be computed as in the previous game. If Y is random and an incorrect password guess is made, then K will be random, since $x_2' \neq 0$ (check is performed in protocol, see Fig. 1) and $pw' - pw_A \neq 0$.

We now have to show that the simulation is sound for any instance of B that generates X_2 and receives a possibly simulated X_1' or α'. If either value is not from A^i, the simulator can extract a witness from the corresponding NIZK proof and check whether the password guess is correct or not. Moreover, if both received values are coming from A^i, the instances are matching otherwise π_α would not be valid. Thus, there is no need to check a password guess in this case.

Since the reduction for the second part of proof in case of relevant server instances is analogous, we get the following bound:

$$\mathbf{Adv}_{P_4}^{ake}(\mathcal{A}) = \mathbf{Adv}_{P_5}^{ake}(\mathcal{A}) + \mathbf{Adv}_{g,\mathbb{G}}^{dsdh}(t') \ . \tag{11}$$

Game G_6 : (Randomize session keys for paired instances) In case of a matching instance, set K randomly (with matching instances holding the same random K).

In this game, in order to bound the difference in the adversary's advantage between $\mathbf{G}_5$ and $\mathbf{G}_6$, we reduce from DTGDH by using a hybrid argument in the following fashion. First, we construct an algorithm $\mathcal{D}$ that for a given tuple $\langle X, Y, Z, DH_g(X,Y), DH_g(X,Z), DH_g(Y,Z), W \rangle$ attempts to break the DTGDH assumption (i.e. determine whether W is random or $W = g^{xyz}$) by running the adversary $\mathcal{A}$ as a subroutine. The algorithm $\mathcal{D}$ chooses two random indexes $i, j \leftarrow \{1, 2, \ldots, n_{se}\}$ and simulates the protocol for $\mathcal{A}$ by computing K randomly for all lexicographically previous instances of (A^i, B^j) that are fully matching, and setting K normally for all lexicographically subsequent instances of (A^i, B^j).

The client instance A^i sets $X_1 = X$, the server instance B^j sets $X_2 = Y$, while the value Z is embedded in place of U. In case of matching instances, A^i sets $\alpha = (DH_g(X,Z)DH_g(X,Y))^{pw_A}$ and B^j sets $\beta = (DH_g(Y,Z)DH_g(X,Y))^{pw_A}$. If the instance is fully matching it computes the shared secret as $K = W^{pw_A}$. If $W = g^{xyz}$ then this simulates computing K normally. If W is random, then K is random since $pw_A \neq 0$.

We now need to check if the simulation is sound for other possible queries to A^i and B^j. If A^i (resp. B^j) receives non-simulated values X_2' and β' (resp. X_1' and α'), it can extract witnesses from the corresponding NIZK and check whether the password guess is correct or not, and respond accordingly. If A^i (resp. B^j) receives an X_2' (resp. X_1') value from the adversary and a simulated β (resp. α), the NIZK proof π_β (resp. π_α) would not be valid due to the labels l_B (resp. l_A). Conversely, if after the first message flow the instances are matching ($X_1 = X_1'$ and $X_2 = X_2'$), but α' or β' are from the adversary, the corresponding password guess will be incorrect.

In case a password guess is correct and the simulation continues, a **Corrupt** query has been made due to $\mathbf{G}_4$. Since the simulator can extract x_1' or x_2' in case of an impersonation attempt, the value K can be computed as the adversary expects it to be. Therefore,

$$\mathbf{Adv}_{P_5}^{ake}(\mathcal{A}) = \mathbf{Adv}_{P_6}^{ake}(\mathcal{A}) + (n_{ex} + n_{se}^2)\mathbf{Adv}_{g,\mathbb{G}}^{dtgdh}(t') \ . \tag{12}$$

Game G_7 : (Randomize α and β) If there was no **Corrupt** query, set α and β randomly in all instances. In the case of a correct password guess to a non-matching instance (after **Corrupt**

query), compute K as the other party would have.

Again the proof is split in two parts. Firstly, using a reduction from DDH, the α values are set randomly. We construct an algorithm $\mathcal{D}$ that for a given tuple $\langle X, Y, Z \rangle$ from a DDH challenger, attempts to break the DDH assumption (i.e. determine whether Z is random or $Z = g^{xy}$) by running the adversary $\mathcal{A}$ as a subroutine. The algorithm $\mathcal{D}$ chooses a random index $i \leftarrow \{1, 2, \ldots, n_{se}\}$, and simulates the protocol for $\mathcal{A}$ by setting an exponent $x_1 pw_A$ (in the computation of α) to be random for all client instances prior to A^i and computing α normally for all client instances after A^i as follows.

The client instance A^i sets $X_1 = X$, while the challenge value Y is embedded in place of U. After receiving $\mathbf{Send}(A^i, (B, X_2', \pi_2'))$, the simulator extracts x_2' to compute $\alpha = Z^{pw_A} X^{x_2' pw_A}$. It is obvious that if Z is random, α is random, too. Also, if $Z = DH_g(X, Y)$, α is computed as in $\mathbf{G}_6$. In case the adversary $\mathcal{A}$ succeeds in the protocol (by guessing the $\mathbf{Test}$ bit b or the correct password), $\mathcal{D}$'s guess to the DDH challenger is $b_1' = 1$, and $b_1' = 0$ otherwise. Note that upon receiving β which corresponds to a correct password guess (either from the adversary or from B), the simulator is able to ensure that other instances of A compute the same key as the adversary or B would, even if α is random. This is possible since the simulator can extract x_2', $x_2' pw'$ values from the received NIZK proofs and check whether the password guess is correct or not, and complete the simulation of A accordingly.

We now have to show that the simulation is sound for any instance of B that generates X_2 and receives possibly simulated X_1' or α' from A^i: 1) if both values are from A^i, set K randomly (due to $\mathbf{G}_6$); 2) if any of two received values is not from A^i, the simulator can extract a witness from the corresponding NIZK proof, check whether the password guess is correct (and satisfy $\mathbf{G}_4$) or not (and set K randomly due to $\mathbf{G}_5$); 3) if an α' that corresponds to a correct password guess submitted to an instance of B is not from A^i and the execution of the protocol continues (a $\mathbf{Corrupt}$ query has been made), the discrete log of X_1 is known and the simulator can compute the shared secret K_B as A would.

The reduction for the second part of proof in case of relevant server instances is analogous, so we get the following bound:

$$\mathbf{Adv}_{P_6}^{ake}(\mathcal{A}) = \mathbf{Adv}_{P_7}^{ake}(\mathcal{A}) + 2n_{se}\mathbf{Adv}_{g,\mathbb{G}}^{ddh}(t') \ . \tag{13}$$

Game $\mathbf{G}_8$: (Randomize sk) Set sk randomly in all instances in which K is set randomly (matching instances get the same sk).

Two games are computationally indistinguishable, since public string e and K are random and H_1 is a computational randomness extractor. This concludes the proof.

$\square$

5   Conclusion

In this paper, we proposed two new variants of J-PAKE, showed that the security proof from [6] can be adapted to cover our proposals, and compared the overall efficiency of all three protocols when instantiated with ECs or SFFs. Since RO-J-PAKE using SFFs is the least efficient because of the implementation of the hash function H_0, it would be interesting to see if it can be proven secure using a large SFF (and therefore, a "small r"), all while using a short-exponent-type

complexity assumption (e.g. as in [29]).

References

[1] BlueKrypt (2015), `http://www.keylength.com/en/4/`

[2] Firefox Sync (2015), `https://www.mozilla.org/en-US/firefox/sync/`

[3] OpenSSL (2015), `https://www.openssl.org/`

[4] Pale Moon (2015), `http://www.palemoon.org`

[5] Thread Protocol (2015), `http://threadgroup.org/`

[6] Abdalla, M., Benhamouda, F., MacKenzie, P.: Security of the J-PAKE Password-Authenticated Key Exchange Protocol. In: 2015 IEEE Symposium on Security and Privacy, SP 2015. pp. 571–587. IEEE Computer Society (2015)

[7] Abdalla, M., Benhamouda, F., Pointcheval, D.: Public-Key Encryption Indistinguishable Under Plaintext-Checkable Attacks. In: Katz, J. (ed.) Public-Key Cryptography – PKC 2015. LNSC, vol. 9020, pp. 332–352. Springer (2015)

[8] Abdalla, M., Fouque, P., Pointcheval, D.: Password-Based Authenticated Key Exchange in the Three-Party Setting. In: Vaudenay, S. (ed.) Public-Key Cryptography – PKC 2005. LNCS, vol. 3386, pp. 65–84. Springer (2005)

[9] Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated Key Exchange Secure Against Dictionary Attacks. In: Advances in Cryptology – EUROCRYPT 2000. LNCS, vol. 1807, pp. 139–155. Springer (2000)

[10] Bellare, M., Rogaway, P.: Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In: ACM Conference on Computer and Communications Security. pp. 62–73. ACM Press (1993)

[11] Bellovin, S.M., Merritt, M.: Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. In: 1992 IEEE Symposium on Research in Security and Privacy, SP 1992. pp. 72–84 (1992)

[12] Boneh, D.: The Decision Diffie-Hellman Problem. In: Buhler, J. (ed.) Algorithmic Number Theory, Third International Symposium, ANTS-III, 1998. LNCS, vol. 1423, pp. 48–63. Springer (1998)

[13] Boyko, V., MacKenzie, P.D., Patel, S.: Provably Secure Password-Authenticated Key Exchange Using Diffie-Hellman. In: Preneel, B. (ed.) Advances in Cryptology – EUROCRYPT 2000. LNCS, vol. 1807, pp. 156–171. Springer (2000)

[14] Brier, E., Coron, J., Icart, T., Madore, D., Randriam, H., Tibouchi, M.: Efficient Indifferentiable Hashing into Ordinary Elliptic Curves. In: Rabin, T. (ed.) Advances in Cryptology - CRYPTO 2010. LNCS, vol. 6223, pp. 237–254. Springer (2010)

[15] Canetti, R., Halevi, S., Katz, J., Lindell, Y., MacKenzie, P.D.: Universally Composable Password-Based Key Exchange. In: Cramer, R. (ed.) Advances in Cryptology – EUROCRYPT 2005. LNCS, vol. 3494, pp. 404–421. Springer (2005)

[16] Goldreich, O., Lindell, Y.: Session-Key Generation Using Human Passwords Only. In: Kilian, J. (ed.) Advances in Cryptology CRYPTO 2001, LNCS, vol. 2139, pp. 408–432. Springer (2001)

[17] Groth, J.: Simulation-Sound NIZK Proofs for a Practical Language and Constant Size Group Signatures. In: Lai, X., Chen, K. (eds.) Advances in Cryptology - ASIACRYPT 2006. LNCS, vol. 4284, pp. 444–459. Springer (2006)

[18] Groth, J., Ostrovsky, R.: Cryptography in the Multi-string Model. In: Menezes, A. (ed.) Advances in Cryptology - CRYPTO 2007. LNCS, vol. 4622, pp. 323–341. Springer (2007)

[19] Hao, F., Ryan, P.: J-PAKE: Authenticated Key Exchange without PKI. Transactions on Computational Science 11, 192–206 (2010)

[20] Harkins, D.: Dragonfly Key Exchange (2015), `https://datatracker.ietf.org/doc/draft-irtf-cfrg-dragonfly/`

[21] Jablon, D.P.: Strong Password-Only Authenticated Key Exchange. ACM SIGCOMM Computer Communication Review 26(5), 5–26 (1996)

[22] Jiang, S., Gong, G.: Password Based Key Exchange with Mutual Authentication. In: Handschuh, H., Hasan, M.A. (eds.) Selected Areas in Cryptography, SAC 2004. LNCS, vol. 3357, pp. 267–279. Springer (2004)

[23] Katz, J., Ostrovsky, R., Yung, M.: Efficient Password-Authenticated Key Exchange Using Human-Memorable Passwords. In: Pfitzmann, B. (ed.) Advances in Cryptology – EUROCRYPT 2001. LNCS, vol. 2045, pp. 475–494. Springer (2001)

[24] Katz, J., Vaikuntanathan, V.: Round-Optimal Password-Based Authenticated Key Exchange. In: Ishai, Y. (ed.) Theory of Cryptography - TCC 2011. LNCS, vol. 6597, pp. 293–310. Springer (2011)

[25] Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: Rabin, T. (ed.) Advances in Cryptology - CRYPTO 2010. LNCS, vol. 6223, pp. 631–648. Springer (2010)

[26] Lancrenon, J., Skrobot, M.: On the Provable Security of the Dragonfly Protocol. In: Lopez, J., Mitchell, C.J. (eds.) Information Security - 18th International Conference, ISC 2015. LNCS, vol. 9290, pp. 244–261. Springer (2015)

[27] MacKenzie, P.: The PAK Suite: Protocols for Password-Authenticated Key Exchange. DIMACS Technical Report 2002-46 (2002)

[28] MacKenzie, P.D.: More Efficient Password-Authenticated Key Exchange. In: Naccache, D. (ed.) Topics in Cryptology - CT-RSA 2001. LNCS, vol. 2020, pp. 361–377. Springer (2001)

[29] MacKenzie, P.D., Patel, S.: Hard Bits of the Discrete Log with Applications to Password Authentication. In: Topics in Cryptology - CT-RSA 2005. LNCS, vol. 3376, pp. 209–226. Springer (2005)

[30] Pointcheval, D.: Password-Based Authenticated Key Exchange. In: Fischlin, M., Buchmann, J.A., Manulis, M. (eds.) Public Key Cryptography - PKC 2012. LNCS, vol. 7293, pp. 390–397. Springer (2012)

[31] Schnorr, C.: Efficient Identification and Signatures for Smart Cards. In: Brassard, G. (ed.) Advances in Cryptology - CRYPTO 1989. LNCS, vol. 435, pp. 239–252. Springer (1989)
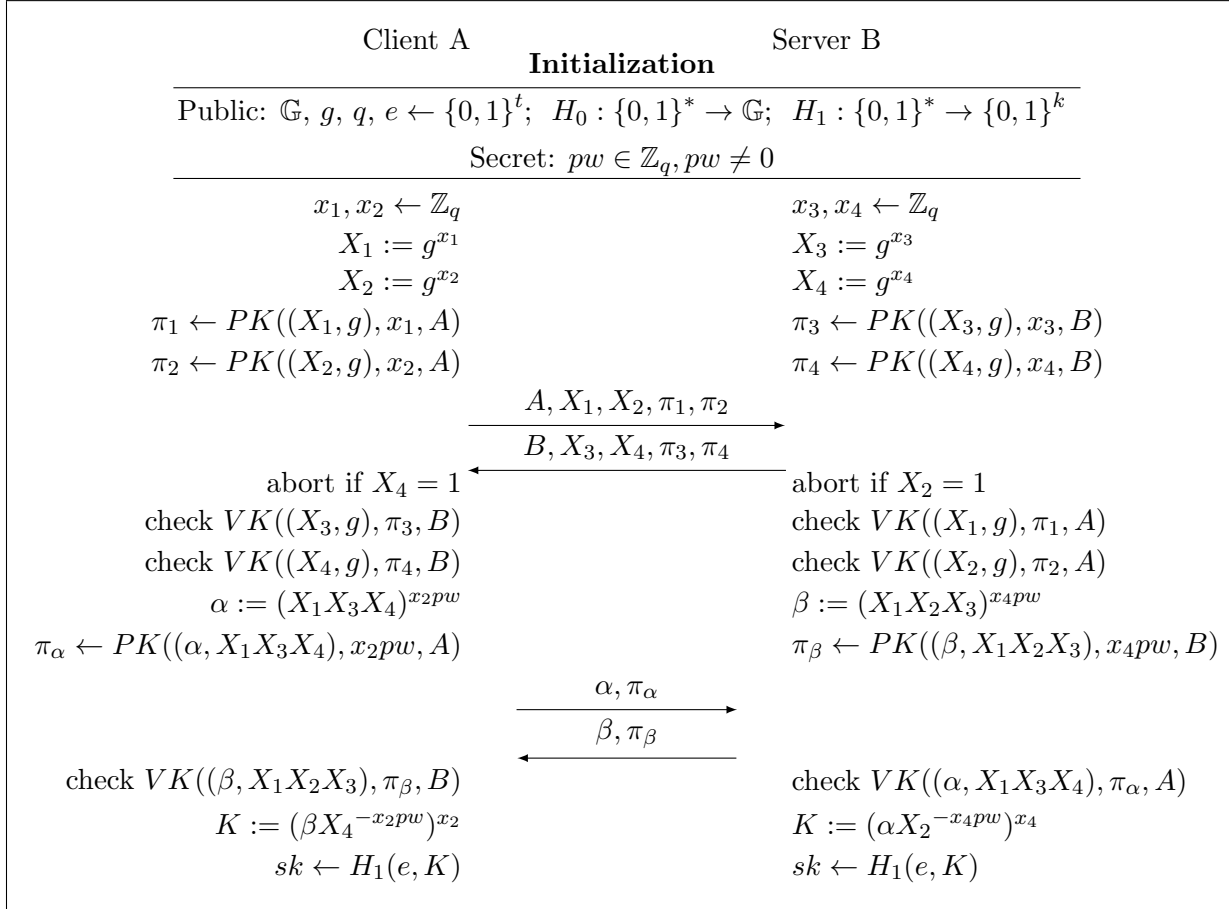
A   Original J-PAKE protocol

Client A | Server B
Initialization

Public: $\mathbb{G}, g, q, e \leftarrow \{0,1\}^t$; $H_0 : \{0,1\}^* \rightarrow \mathbb{G}$; $H_1 : \{0,1\}^* \rightarrow \{0,1\}^k$

Secret: $pw \in \mathbb{Z}_q, pw \neq 0$

$$x_1, x_2 \leftarrow \mathbb{Z}_q \qquad\qquad x_3, x_4 \leftarrow \mathbb{Z}_q$$
$$X_1 := g^{x_1} \qquad\qquad X_3 := g^{x_3}$$
$$X_2 := g^{x_2} \qquad\qquad X_4 := g^{x_4}$$
$$\pi_1 \leftarrow PK((X_1, g), x_1, A) \qquad\qquad \pi_3 \leftarrow PK((X_3, g), x_3, B)$$
$$\pi_2 \leftarrow PK((X_2, g), x_2, A) \qquad\qquad \pi_4 \leftarrow PK((X_4, g), x_4, B)$$

$$\xrightarrow{\quad A, X_1, X_2, \pi_1, \pi_2 \quad}$$
$$\xleftarrow{\quad B, X_3, X_4, \pi_3, \pi_4 \quad}$$

abort if $X_4 = 1$ | abort if $X_2 = 1$
check $VK((X_3, g), \pi_3, B)$ | check $VK((X_1, g), \pi_1, A)$
check $VK((X_4, g), \pi_4, B)$ | check $VK((X_2, g), \pi_2, A)$
$\alpha := (X_1 X_3 X_4)^{x_2 pw}$ | $\beta := (X_1 X_2 X_3)^{x_4 pw}$
$\pi_\alpha \leftarrow PK((\alpha, X_1 X_3 X_4), x_2 pw, A)$ | $\pi_\beta \leftarrow PK((\beta, X_1 X_2 X_3), x_4 pw, B)$

$$\xrightarrow{\quad \alpha, \pi_\alpha \quad}$$
$$\xleftarrow{\quad \beta, \pi_\beta \quad}$$

check $VK((\beta, X_1 X_2 X_3), \pi_\beta, B)$ | check $VK((\alpha, X_1 X_3 X_4), \pi_\alpha, A)$
$K := (\beta X_4^{-x_2 pw})^{x_2}$ | $K := (\alpha X_2^{-x_4 pw})^{x_4}$
$sk \leftarrow H_1(e, K)$ | $sk \leftarrow H_1(e, K)$

Figure 5: The J-PAKE protocol. The public random value e is added as in Abdalla et al. to remove unnecessary random oracle assumption.

B   Security proofs

B.1   Security Proof of RO-J-PAKE with Partial Labels

Here we prove the security of RO-J-PAKE protocol, even in case that labels l_A and l_B only contain an identity of the originator of NIZK proofs π_α and π_β, as in originally proposed J-PAKE. In this case, the major difference occurs in the game $\mathbf{G}_3$ where we need to show that the adversary can not manipulate X_1 and X_2 values such that non-matching client and server instances compute the same bases for α or β. Also, without full labels, we need to use a hybrid argument in $\mathbf{G}_5$ to reduce to DSDH assumption.

Theorem 3. *Consider **RO-J-PAKE** as specified in Fig. 1 with labels l_A and l_B only containing the identity A and B respectively. Let $\mathcal{A}$ be an adversary that runs in time at most t, and makes at most n_{se}, n_{ex}, n_{re}, n_{te}, n_{h0} queries of type **Send**, **Execute**, **Reveal**, **Test** and RO queries to*

H_0. *It holds that*

$$\boldsymbol{Adv}^{ake}_{\mathtt{ro-j-pake}}(\mathcal{A}) \leq \frac{n_{se}}{N} \;+\; O\Bigg(\frac{(n_{se} + n_{ex} + n_{ho})^2}{q} \;+\; \frac{n_{h0}^2}{q} \;+\; n_{se}\boldsymbol{Adv}^{dsdh}_{g,\mathbb{G}}(t')$$
$$+\, (n_{ex} + n_{se}^2)\boldsymbol{Adv}^{dtgdh}_{g,\mathbb{G}}(t') \;+\; 2n_{h0}n_{se}\boldsymbol{Adv}^{ddh}_{g,\mathbb{G}}(t')$$
$$+\, (n_{re} + n_{te})\boldsymbol{Adv}^{comp}_{ext_R}(t') \;+\; \boldsymbol{Adv}^{uzk}_{NIZK}(t') \;+\; \boldsymbol{Adv}^{ext}_{NIZK}(t')\Bigg),$$

and where $t' = O(t + (n_{se} + n_{ex} + n_{ho})t_{exp})$ with t_{exp} being the time required for an exponentiation in $\mathbb{G}$.

Game G_3 : (Disallow same-base attacks) In case of the client instance A^i that generate X_1 and the server instance B^j that generate X_2, the following events are avoided:

1. If $X_2 \neq X'_2$ after $\mathcal{A}$ has made a **Send**$(A^i, (B, X'_2, \pi'_2))$ query, and A^i and B^j compute the same base for α.

In this case, the values $D_A = H_0(A, B, X_1, X'_2)$ and $D_B = H_0(A, B, X'_1, X_2)$ are distinct (even if $X_1 = X'_1$), since we avoid collisions on H_0 in G_2. The client will compute a base for α as $T^\alpha_A = D_A X'_2$, while the server will compute the same base as $T^\alpha_B = D_B X_2$. Remember that π_α contains T^α_A, which server must check before verifying the validity of π_α. The probability that $T^\alpha_A = T^\alpha_B$, or more precisely, that the output of a hash function $H_0(A, B, X_1, X'_2)$ is equal to the bad value $D_B X_2 / X'_2$ is n_{ho}/q. Notice also that in sub-case when $X_1 = X'_1$, the base for β computed by the client $T^\beta_A = D_A X_1$ can not be equal to the same base $T^\beta_B = D_B X_1$ computed by the server – also due to G_2.

2. If $X_1 \neq X'_1$ after $\mathcal{A}$ has made a **Send**$(B^j, (A, X'_1, \pi'_1))$ query, and A^i and B^j compute the same base for β.

As in case above, the output of a hash function $H_0(A, B, X'_1, X_2)$ is equal to the bad value $D_A X_1 / X'_1$ with the probability of n_{ho}/q. Notice that in sub-case when $X_2 = X'_2$, $t^\alpha_A \neq t^\alpha_B$ since this would mean that $D_A = D_B$ for distinct inputs, which is avoided in G_2.

3. If $X_1 \neq X'_1$ and $X_2 \neq X'_2$ after $\mathcal{A}$ has made a **Send**$(B^j, (A, X'_1, \pi'_1))$ and a **Send**$(A^i, (B, X'_2, \pi'_2))$, such that $T^\alpha_A = T^\alpha_B$ or $T^\beta_A = T^\beta_B$.

The third case is practically covered with previous two, and the analysis is analogous. In the case either of the above three events occur, the protocol halts and $\mathcal{A}$ fails.

$$\mathbf{Adv}^{ake}_{\mathrm{P}_2}(\mathcal{A}) = \mathbf{Adv}^{ake}_{\mathrm{P}_3}(\mathcal{A}) + O\left(\frac{n_{h0}}{q}\right) \ . \tag{14}$$

Game G_5 : (Randomize session keys for wrong password guesses) In case of an incorrect password guess to a non-matching instance, K is set randomly.

In this game, the reduction is very close to the one in [6]; we use similar hybrid argument, and split the proof in two parts. At first we set K randomly only in the non-matching client instances in case of a wrong password guess – we will call those *target* client instances.

We construct an algorithm $\mathcal{D}$ that given a tuple $\langle X, Y \rangle$, where $X \leftarrow g^x$ and $Y \in \mathbb{G}$, attempts to break DSDH assumption (i.e. determine whether Y is random or $Y = g^{x^2}$) by running the adversary $\mathcal{A}$ as a subroutine. The algorithm $\mathcal{D}$ chooses a random index $i \leftarrow \{1, 2, \ldots, n_{se}\}$ and simulates the protocol for $\mathcal{A}$ by setting K randomly for all target client instances prior to A^i, and computing K normally for all client instances after A^i as follows.

For a given DSDH instance $\langle X, Y \rangle$, the client instance A^i sets $X_1 = X$. When A^i receives $\mathbf{Send}(A^i, (B, X_2', \pi_2'))$ query, the simulator extracts x_2' from valid π_2', computes D_A and sets $\alpha = X^{(d_A + x_2')pw_A}$. After receiving $\mathbf{Send}(A^i, (\beta', \pi_\beta'))$ and extracting a witness from π_β', the client instance A^i computes $K = (\beta' g^{-x_2' x pw_A})^x = X^{d' pw' x_2'} Y^{x_2'(pw' - pw_A)}$. Note that the value pw' is obtained by dividing extracted witnesses from the NIZK proofs $pw' = Ext(\pi_\beta')/Ext(\pi_2')$, while d' comes from the list $\mathcal{L}_{h0}$ (see Fig. 3). In case of matching instances or a correct password guess $K = X^{d_A pw_A x_2'}$, since pw' is equal to pw_A. If Y is real DSDH challenge, then the value K will be computed as in $\mathbf{G}_4$. On the other hand, if Y is random and an incorrect password guess is made, then K will be random, since $x_2' \neq 0$ (check is performed in protocol, see Fig. 1) and $pw' - pw_A \neq 0$.

We now have to show that simulation is sound for any instance of B that generates X_2 and receives possibly simulated X_1' or α'. If any of two received values is not coming from A^i, the simulator can extract witness from corresponding NIZK proof and check whether password guess is correct or not. Moreover, if both received values are coming from A^i and instances are non-matching, π_α would be valid only if non-matching A^i and B^j would compute the same base for α – meaning that $D_A^i X_2' = D_B^j X_2$. However, this event is discarded in the $\mathbf{G}_3$, and therefore there is no need to check password guess in this case.

Since the reduction for the second part of proof in case of relevant server instances is analogous, we get the following bound:

$$\mathbf{Adv}_{\mathrm{P}_4}^{ake}(\mathcal{A}) = \mathbf{Adv}_{\mathrm{P}_5}^{ake}(\mathcal{A}) + n_{se}\mathbf{Adv}_{g,\mathbb{G}}^{dsdh}(t') \ . \tag{15}$$

C   Experiment Source Codes

```
//   ./tests/TestAddMul_x 80 23 7
#define N_TESTS 1
//#define DEBUG 1

//stringstream
#include <sstream>

#include "Plaintext.h"
#include "DoubleCRT.h"
#include "Ciphertext.h"
#include "NTL/ZZ_pX.h"
#include <NTL/vector.h>

#include "SHA1.hpp"

#include <time.h>
#include "FHE-SI.h"
#include <vector>

//time cost
#include <ctime>
std::clock_t start;
double duration;
```

```cpp
//ns time test
#include <chrono>
typedef std::chrono::high_resolution_clock Clock;

#include <NTL/ZZ_p.h>
#include <NTL/ZZ.h>
//http://www.shoup.net/ntl/doc/ZZ.cpp.html
//http://www.shoup.net/ntl/doc/ZZ_p.cpp.html

int len=256;
int TIMES=1000;

ZZ p;
ZZ_p a, b;
ZZ_p Px, Py;

void CurveGen(){
        //ZZ p,
        //ZZ_p a, b;

        //gen p
        GenPrime(p, len, 80); //prime with 2^(-80)
        while(rem(p,3)!=2)
                GenPrime(p, len, 80);
        //cout<<"p:"<<hex<<p<<endl;

        ZZ_p::init(p);

        //gen a, b
        random(a);
        random(b);

        //gen a random point, not neccecarily on the curve
        //From P12 "Generation Methods of Elliptic Curves"
        //the whole computation of G and verifying its order
    //is of bit-complexity O(log^4(p))
        random(Px);
        random(Py);
}

void Pgen1(ZZ m){
        ZZ tp3;
        ZZ_p u, v, x, y;
        ZZ_p tp1, tp2, tp4;

        //u=hash(m)
        //sha-1
        //https://github.com/jasinb/sha1/blob/master/main.c
        conv(u,m);

        //embed message
        //compute v
        mul(tp1,a,3);
        power(tp2, u, 4);
        sub(tp1,tp1,tp2);
        mul(tp2, u, 6);
        div(v, tp1, tp2);

        power(tp1, v, 2);
        sub(tp1, tp1, b);
        mul(tp2, u, u); // u^6/27=(u^2/3)^3
        div(tp2, tp2, 3); //tp2=u^2/3
        power(tp4, tp2, 3);
        sub(tp1, tp1, tp4);
```

```
        mul(tp3, p, 2);
        sub(tp3, tp3, 1);
        div(tp3, tp3, 3);

        power(tp1, tp1, tp3);
        add(x, tp1, tp2);   //x

        mul(tp1, u, x);
        add(y, tp1, v);   //y

        //cout<<"x:"<<hex<<x<<endl;
        //cout<<"y:"<<hex<<y<<endl;
}

//point addition in Jacobian
void AD(ZZ_p& X1,ZZ_p& Y1,ZZ_p& Z1,ZZ_p X2,ZZ_p Y2){
        ZZ_p A, B, C, D, E, F, G;
        ZZ_p Z2;

        conv(Z2,1);

        A=(X1*power(Z2,2));
        B=(X2*power(Z1,2));
        C=(Y1*power(Z2,3));
        D=(Y2*power(Z1,3));
        E=(B-A);
        F=(D-C);

        G=power(Z2,2);

        X1=(-power(E,3)-2*A*power(E,2)+power(F,2))  ;
        Y1=(-C*power(E,3)+F*(A*power(E,2)-X1))  ;
        Z1=(Z1*Z2*E)  ;
}

//point doubling in Jacobian
void DB(ZZ_p& X1,ZZ_p& Y1,ZZ_p& Z1){
        ZZ_p A, B;

        A=(4*X1*power(Y1,2))  ;
        B=(3*power(X1,2)+a*power(Z1,4))  ;

        Z1=(2*Y1*Z1)  ;
        X1=(-2*A+power(B,2))  ;
        Y1=(-8*power(Y1,4)+B*(A-X1))  ;
}

//using m*P for point hashing
//using Jacobian coordination
//if sha-1, m is 160-bit?
void Pgen2(ZZ m){
        ZZ_p X1, Y1, Z1;
        ZZ_p tp1;

        X1=Px;//suppose MSB of m=1
        Y1=Py;
        conv(Z1, 1);
        for(int i=len-1; i>=0; i--)
        {
                DB(X1, Y1, Z1);
                if(bit(m,0))  //LSB
                        AD(X1,Y1,Z1,Px,Py);
                m=m>>1;
        }
```

```
        tp1=inv(power(Z1,3));
        Y1=Y1*tp1;
        X1=X1*Z1*tp1;

        //cout<<"X1:"<<hex<<X1<<endl;
        //cout<<"Y1:"<<hex<<Y1<<endl;
}


ZZ stringToNumber(string str)
{
    ZZ number = conv<ZZ>(str[0]);
    long len = str.length();
    for(long i = 1; i < len; i++)
    {
        number *= 16;
        number += conv<ZZ>(str[i]);
    }

    return number;
}

int main(int argc, char *argv[]) {
  ZZ tpm;
  vector<ZZ> m;
  SetSeed(to_ZZ(time(0)));

  //Curve Gen
  CurveGen();

  //Message Gen1
  for (int i = 0; i < TIMES; i++)
  {
        RandomBits(tpm, len);
        m.push_back(tpm);
  }


  //hash 1
  //"How to Hash into Elliptic Curves"
  start = std::clock();

  for (int i = 0; i < TIMES; i++)
  {
        Pgen1(m.back());
        m.pop_back();
  }

  duration = ( std::clock() - start ) / (double) CLOCKS_PER_SEC;
  cout<<"HASH1:"<< duration/TIMES <<'\n';

  //Message Gen2
  for (int i = 0; i < TIMES; i++)
  {
        RandomBits(tpm, len);
        m.push_back(tpm);
  }

  //hash 2
  //m*P
  start = std::clock();

  for (int i = 0; i < TIMES; i++)
  {
        Pgen2(m.back());
```

```
        m.pop_back();
    }

    duration = ( std::clock() - start ) / (double) CLOCKS_PER_SEC;
    cout<<"HASH2:"<< duration/TIMES <<'\n';
}
```