

# AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT

Björn Haase and Benoît Labrique

Endress+Hauser Conducta GmbH&Co. KG, Germany

[bjjoern.haase@endress.com](mailto:bjjoern.haase@endress.com)

## Abstract.

Increasingly connectivity becomes integrated in products and devices that previously operated in a stand-alone setting. This observation holds for many consumer applications in the so-called "Internet of Things" (IoT) as well as for corresponding industry applications (IIoT), such as industrial process sensors. Often the only practicable means for authentication of human users is a password. The security of password-based authentication schemes frequently forms the weakest point of the security infrastructure.

Missing integration of IoT or IIoT device in a WEB-PKI should be considered a significant real-world risk. In this setting, verifier-based password-authenticated key-exchange (V-PAKE) protocols are known to provide a significant security improvement by preventing phishing and offline dictionary attacks.

For IIoT, availability concerns for the case of failures of (part of) the communication infrastructure makes local storage of access credentials mandatory. The larger threat of physical attacks makes it important to use memory-hard password hashing.

This paper presents a corresponding tailored protocol, AuCPace, together with a security proof within the Universal Composability (UC) framework considering fully adaptive adversaries. AuCPace uses CPace as a building block which could be used as a stand-alone balanced PAKE protocol. Moreover, we show how AuCPace could optionally provide for pre-computation attack resistance. In this paper we also introduce a new security notion of partially augmented PAKE that provides specific performance advantages for constrained servers.

We also present an actual instantiation of our protocol, AuCPace25519, and present performance results on ARM Cortex-M0 and Cortex-M4 microcontrollers, demonstrating the suitability of AuCPace for the constrained server setting.

*This specific paper revision is an update of the journal version. It was setup for the PAKE selection process of the CFRG working group of the IETF for which AuCPace and CPace have been nominated.*

**Keywords:** Password Authenticated Key Exchange, V-PAKE, PAKE, elliptic curves, Cryptographic Protocols, Universal Composability, IEC-62443, Industrial Control, Curve25519, X25519, OPRF

## 1 Introduction

Since recently, wireless and networking technology becomes integrated in products and devices that previously operated in a stand-alone setting, both in consumer applications in the so-called "Internet of Things" (IoT) as well as in the corresponding industry setting, the "Industrial IoT" (IIoT). Often communication technology and security protocols are employed that were not originally tailored and designed for the resource-constrained setting and the specific threat model.

In comparison to conventional un-connected devices, security becomes a crucial aspect to consider, specifically in the IIoT. Often, users tend to use short and easily memorable passwords. For this reason emerging industry standards, such as from the IEC-62443 family, rightfully require two-factor authentication for higher security levels (SL-3, SL-4). Still then, however, suitable protection of the second factor "password" remains important.

Today, in most Internet communication and many IIoT applications protocols such as TLS based web servers are used, that were not originally designed for the IIoT use-case. Mostly, a trustworthy and failure-free Public-Key-Infrastructure (PKI) is indispensable for providing even basic protection of passwords, e.g., against phishing or man-in-the-middle attacks. However in the IIoT setting, specifically for industrial installations not fully operating according to standards such as IEC-62443, today integration of the devices in a PKI is not always available.

## 1.1 Note on the current standardization activities for industrial installations

Security is a new topic for most industry installations and component suppliers. For the project of securing industrial installations, suppliers and implementers currently seem to first concentrate on standardizing the security protocols of the machine-machine interfaces. User authentication tends to be addressed in a second step only.

One example is the Common Industrial Protocol (CIP Security) standard family which forms the base for the EtherNet/IP(tm) and DeviceNet(tm) standards. Here, while the standardization of security mechanisms for the machine-machine interfaces is widely settled, the standardization of user authentication as part of the "CIP Authorization Profile" is still a work in progress.

In our assessment, we consider user authentication to be as critical as machine-machine authentication. In many settings the attacker is given equivalent power over an installation by gaining control over human machine interfaces. With this paper we also aim at contributing to the project of resolving this issue.

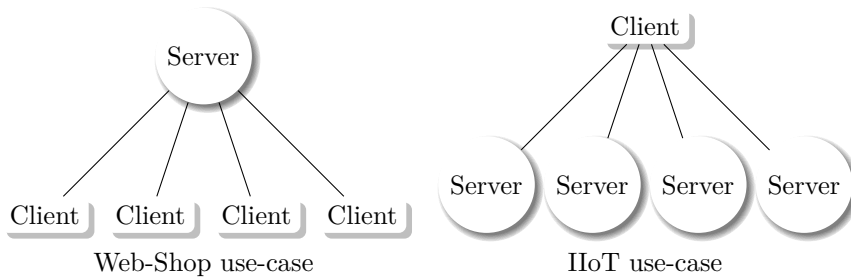
## 1.2 Differences between conventional web security and IIoT

One of the most remarkable differences between the conventional web server setting (here referred to as "web shop" use case) and the typical IIoT setting corresponding, e.g., to an industrial plant is illustrated in Figure 1. In the former case few servers, e.g., web servers, interface to many clients, e.g., web browsers which come with pre-installed configuration for certificate authorities trusted by the browser supplier. In the latter case, one single client, e.g., a tablet-computer based human machine interface (HMI), might be used for configuring many servers, e.g., sensors or control valves.

Unlike in the "web shop" use case, in the IIoT setting a dramatically larger number of server certificates needs to be configured and maintained. Often self-signed certificates are used for servers, leading to the significant risk that the essential corresponding configurations on the client side (e.g., browsers) are omitted. Operators and system administrators used to the more convenient "web shop" setting might not even be aware that such configuration is mandatory for security.

As a consequence, the threat of PKI failures should be considered very carefully for remote HMI access to (I)IoT units. This is one of the reasons, why since recently strategies for password protection as a hedge against PKI failures have regained academic and industrial interest [JKX18, PW17, HL17].

Moreover many IIoT devices, notably battery-driven devices, will not be permanently "online". Availability concerns for the case of (partial) failure of networking infrastructure often make it mandatory to locally store user credentials, typically in unprotected memory.



**Figure 1:** Use-cases for conventional Internet applications and IIoT.

Furthermore, IoT devices might be much more exposed to physical attacks. For all of these reasons, protection of passwords forms a crucial point of any (I)IoT security solution.

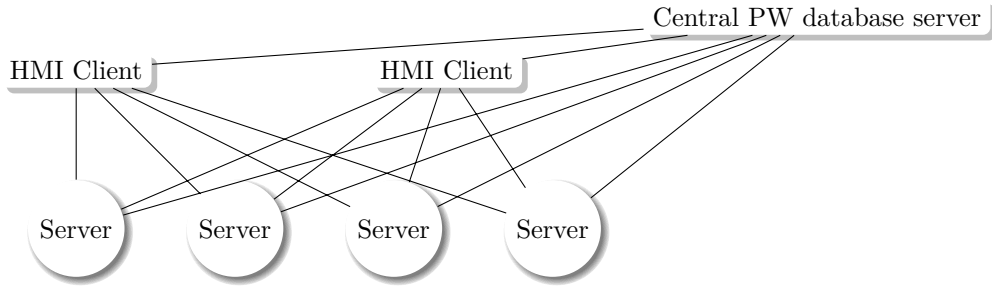
### 1.3 Strategies for protecting passwords

For the protection of passwords two complementary approaches could be distinguished. On the one hand, memory-hard password hash algorithms such as scrypt [PJ12] and Argon2 [BDKJ16] aim at increasing the cost of offline dictionary attacks. On the other hand password-based key exchange (PAKE) protocols allow for establishing a secure, high-entropy shared session key over an insecure communication channel. This holds even if only a low-entropy secret key, the password  $pw$  is shared (see, e.g., [Gre18] for a recent gentle introduction to the concept of PAKE).

One of the important advantages of PAKE protocols published since the early works by Bellare and Merritt and Jablon [BM92, Jab96] is the fact that neither a public-key infrastructure (PKI) nor a trusted hardware component capable of securely storing high-entropy keys is required as prerequisite. PAKE protocols, thus, match very closely the needs of the IIoT use-case.

PAKE protocols essentially come in two variants. Firstly, so-called balanced or symmetric PAKE protocols, for instance [BM92, CHK<sup>+</sup>05, BFK09, Jab96, BMP00], are designed such that both parties, initiator and responder, require that the same password  $pw$  is available on both sides. Secondly, so-called verifier-based PAKE protocols (also known as V-PAKE, asymmetric or augmented PAKE) protocols could be distinguished, where the server is given access only to a password-verifier  $W$  and the clear-text password is only available to the client party, e.g., [Wu98, PW17, GMR06, Jab97]. V-PAKE allows for some limited additional protection for the case that the attacker gets access to the server's password-verifier database. The additional protection due to augmentation should not be over-estimated but still could make the task significantly more difficult for the attacker. In all known protocols V-PAKE comes with significant additional computational overhead in comparison to PAKE.

A peculiarity in the industrial IIoT setting is that we should be expecting to find the very same password in use on many small devices. Due to password re-use the compromise of one small server might affect the security of a larger infrastructure. On the other hand, availability concerns for the case of failures of (part of) the network infrastructure often make it mandatory to locally store this sensitive data and often no protected storage media, such as smart-card circuits, are available. For this reason verifier-based PAKE using memory hard password hashing usually should be considered to provide the best practically feasible security strategy regarding HMI authentication for IIoT applications.



**Figure 2:** IIoT setting with central password distribution infrastructure.

#### 1.4 Why industrial instrumentation needs a specially tailored V-PAKE protocol

Unfortunately, it is not uncommon that IoT and industrial devices have only very limited energy and computational resources available, specifically if they have to be conformable with the constraint set that applies for intrinsically safe explosion protection (IEC60079-11) [HL17].

For important classes of devices, use of memory-hard password hashing is precluded due to the limited memory and computational capacities. Also the computational complexity of some established V-PAKE protocols that were originally developed for office information technology might prevent actual use in the IIoT. Note that all of the most efficient known augmented protocols (requiring three exponentiations) or components of the protocols, such as AugPake [SKI10] or OPAQUE [JKX18] are covered by patents. It is important to consider that also devices even smaller than typical IIoT devices need sound password protection, such as, e.g., legacy fieldbus systems with bluetooth-based wireless HMI interfaces [HL17].

In all of these settings, efficiency considerations and pending patents will be a crucial factor ultimately deciding upon whether or not sound password protection could actually be implemented by manufacturers. Note that specifically for the smallest devices such as, e.g., temperature sensors, a significant commercial cost constraint applies. Power efficiency, code size and ease-of-implementation are crucial factors that will be decisive for actual roll-out of a more secure or insecure solution, i.e., efficiency is of utmost importance.

Regarding efficiency of V-PAKE solutions, optimizations need to apply on all levels of the security implementation, protocol design, algorithmic optimizations (e.g., group operations on elliptic curves) and low-level arithmetic. Unfortunately, most protocols today were mainly designed with typical office environments in mind.

It is also important to consider that authentication protocols are often embedded in a larger system, involving, e.g., protocols allowing for distributing password verifiers to server entities from a central management infrastructure as sketched in figure 2. Note that in the IEC-62443-4-2 draft standard, centralized user account management systems are declared mandatory even at the lowest security level SL-1. This provides significant challenges for all devices that are not permanently "on-line" or isolated by purpose from the main company network, i.e., in settings where a direct network connection to a central password database server may be not reliably available.

#### 1.5 Contribution of this paper

This paper aims at contributing to the project of securing IIoT applications by comprehensively addressing all of efficiency, implementation and patent issues. We aim at doing so by introducing an efficiency-optimized V-PAKE protocol "Augmented Composable Password Authenticated Connection Establishment" (AuCPace) and an actual instantiation AuC-

Pace25519 that is, to the best of our knowledge, freely usable without inflicting intellectual property rights and specifically tailored for the exact subset of devices where security is most likely to be discarded: Small extremely resource-constrained and/or low-cost devices where integrating sound security is particularly difficult.

The project of protecting industry installations clearly requires considering a larger scope, such as sketched in figure 2. For the scope of this work, however, we identified the need to concentrate in a first step on the sub-component of the authentication protocol running between the HMI server and client units, i.e., leaving out the significant complexity of the user credential distribution process for local password registration.

Our scheme arranges for use of memory-hard password hashes also on smallest devices with little memory, since it is deferring the costly password hashing to the client entities.

Our work builds up upon the work in [HL17] by explicitly providing formal security proofs for the specific optimized design choices therein, e.g., regarding the cheaper alternative to conventional point verification employing twist security [BL19] and avoidance of costly and memory-consuming point compression. Note that such aspects are usually not relevant for the conventional office IT setting.

We do so by providing a security proof for AuCPace in the UC framework with joint state [CR03]. We base our analysis on the UC-based security model of [GMR06]. We show that our protocol, unlike most Diffie-Hellman-based constructions, could be proven secure in the UC model, even when considering fully adaptive adversaries. We extend this model for allowing a new operation mode that we coined *partially augmented* PAKE. We show how it is possible to implement AuCPace in its partially augmented variant without any computational overhead in comparison to conventional "balanced" PAKE for the resource-constrained server, while maintaining all of the most relevant security guarantees in the IIoT setting.

Our protocol could be clearly modularized into 1.) a balanced sub-protocol coined CPace (Composable Pace) which shares important design features with PACE [BFK09] and 2.) an augmentation layer allowing for both conventional augmentation and our newly introduced partial augmentation.

As one concrete instantiation we present a protocol AuCPace25519 which adds further optimizations also on the group arithmetic and field arithmetic level. We present performance results for both, AuCPace and Diffie-Hellman protocols for ARM Cortex M0 and Cortex M4 microcontrollers.

We hope that the new speed-records for constant-time implementation of both, PAKE and the X25519 Diffie-Hellman Protocol on the ARM Cortex M4, that we report in this paper will make it possible to enlarge the set of targets that could afford integrating state-of-the-art security technology.

## 1.6 Organization of this paper

For this paper we consider different groups of readers being more or less inclined to theoretical analysis and actual implementation on constrained targets respectively. For this reason, this paper is organized as follows.

In section 2 we first review related work on PAKE protocols and the definitions of security used for their respective analysis. Since AuCPace may be considered a SPEKE [Jab96] variant, we will explain, why we chose SPEKE as basis despite the fact that for SPEKE no proof is available, when used with elliptic curves. We then give a short introduction to the concepts and methods used for proofs in the framework of Universal Composability (UC). In section 2 we also will give a short introduction to the concept of "partial augmentation" which will be introduced more formally only later as part of the security proofs. We do so in order to make the paper self-contained also for readers willing to skip the security proofs in a first step.

Subsequently, in section 3 we will introduce the full protocol AuCPace. There we will also explain the design guidelines that motivated specific choices and constructions.

We then present the proofs, first handling the case of the balanced component CPace (section 4). In section 5 we then use the composition theorem for proving security of our V-PAKE construction. In the subsequent section 6 we introduce the concept of "partial augmentation" and show how to essentially halve the server's computational complexity. In section 7 we compare our proposal AuCPace with other efficient PAKE constructions.

Subsequently in section 8 we describe the implementation strategy on Cortex M0 and M4 microcontrollers for our reference implementation AuCPace25519. We conclude the paper by presenting actual performance benchmarks on different microcontrollers and embedded bluetooth-transceiver platforms in section 9.

Major parts of the optimized low-level code presented in this paper will be made freely available under a CC0 license at [HL18].

In the appendix we present an outline how the very similar balanced protocol from [HL17] could be proven secure in the UC model.

**Update July 2019:** This paper version is an update of the official TCHES version. It was setup for a current process of the CFRG working group of the IETF. AuCPace has been nominated for the PAKE selection process. This update adds the specification of the "strong AuCPace" protocol in appendix C and our official responses regarding the formal questions asked within the CFRG selection process in appendix D.

In appendix E we give an overview of the different options of the AuCPace protocol family and the corresponding cryptographic assumptions and security guarantees (ideal functionalities).

## 2 Review of PAKE protocols and their security analysis

### 2.1 Overview on PAKE protocols

Despite the clear-cut security advantages, there are a couple of reasons that hampered use of PAKE protocols in a number of applications [EKSS09]. Here patent pitfalls did play a major role. Notably, EKE [BM92] and SPEKE [Jab96] were patented until very recently and also some of the most efficient protocols in the IEEE 1363 standard family are patented. This resulted in protocols such as SRP[Wu98], J-PAKE [HR10] and PACE [BFK09] which did include additional complexity solely for patent circumvention.

As a result of patent circumvention, a large number of different PAKE protocols have been presented. This did hamper thorough security analysis. Firstly proofs became more complex or impractical because of the additional complexity for patent circumvention. Secondly the number of protocol variants grew, reducing the amount of analysis effort spent on each candidate.

Since the SPEKE patent has expired recently, some of the circumventions used for PACE [BFK09] became obsolete. CPace, as presented in this paper, has been developed in the process of removing patent circumvention steps. We did observe that the resulting changes allowed for a natural way of agreeing on a session id *before* entering the protocol and thus opened a path for a proof strategy within the UC framework.

Many protocols, including EKE and SPEKE, have been first suggested without a formal security proof. Doing so comes at the risk of accidentally including serious design-flaws. Many PAKE protocols have later been shown to be insecure. A recent example showing the need of thorough security analysis is the case of zkPAKE presented in [MRA15] that has been shown vulnerable to offline guessing attacks in [BRSS18].

For a recent comprehensive overview over different PAKE protocols and proof strategies see, e.g., [SOAA15].



## 2.2 Security guarantees of PAKE and V-PAKE protocols

All types of PAKE and V-PAKE protocols base security on a low-entropy secret, the password. The direct consequence is that they could not unconditionally be proven secure because in any case exhaustive or dictionary searches by so-called "online attacks" could not be prevented. For each "online" session with a server an attacker always could test at least one password. Due to the low entropy this attack should always be assumed to succeed.

Online attacks could however be mitigated by rate-limiting countermeasures, e.g., by wait times after observing failed login attempts. This countermeasure could not be applied if the authentication protocol allows for "offline" password tests, as, e.g., in case of weak challenge-response protocols.

The security models for PAKE and V-PAKE, thus aim at proving that the "online attack" is the best available attack strategy and that at most one password per online session could be tested by the adversary.

V-PAKE goes one step further than balanced PAKE by additionally considering the case that an attacker might get access to the password database of a server by "compromising" it. This type of attack should be considered highly relevant for (I)IoT devices because often the password database is stored in unprotected memory. Unlike in a "server-room setting" an attacker might more easily gain physical access.

In the event of a server compromise it is technically infeasible to prevent offline attacks. Here V-PAKE protocols aim at forcing the adversary to additionally mount an offline search, i.e., the attacker is required to successfully mount both, server compromise and offline dictionary attacks for a successful impersonation.

As a consequence a real security improvement could actually be realized only if the V-PAKE protocol is used in conjunction with dedicated (specifically with memory-hard) password hashes that slow down an offline search.

For most augmented PAKE protocols, it is possible to interleave the substep of offline dictionary attack with the substep of server compromise, e.g., by pre-computing so-called rainbow tables. In [JKX18] Jarecki, Krawczyk and Xu have introduced the notion of "pre-computation attack resistance" of a so-called *strong* V-PAKE protocol that allows an attacker to mount the offline search only *after* having successfully compromised the server.

In this paper we additionally introduce the notion of a "partially augmented" PAKE protocol. This notion becomes meaningful in a setting where several servers share the same password database. While for the compromised server, a partially augmented protocol provides guarantees similar to a balanced PAKE protocol, it preserves the security guarantees of a V-PAKE protocol for the remaining subset of servers.

## 2.3 Security models

One important approach for analyzing a PAKE is the game-based "real-or-random" (ROR) model by Bellare, Pointcheval and Rogaway (BPR) [BPR00]. This was later extended to the so-called "find-then-guess" (FTG) model [AFP05]. Simulation-based proof techniques, were established, e.g., by Boyko, MacKenzie and Patel (BMP) in [BPR00].

In 2005 Canetti, Halevi, Katz, Lindell and MacKenzie [CHK<sup>+</sup>05] have suggested an alternative approach, based on the framework of Universal Composability (UC) [Can01], specifically in its joint-state version [CR03]. It has been shown that UC-secure PAKE constructions could not be realized without either, idealized assumptions, such as random oracles or a common reference string [CHK<sup>+</sup>05]. One of the advantages of analyzing PAKE protocols in the UC framework is that no assumptions regarding the password distribution apply. Related passwords or mistyped related passwords and forward secrecy are inherently also considered. For this reason, the UC-based approach is considered to be providing particularly strong security guarantees [PW17, JKX18].

## 2.4 Review of SPEKE and SPEKE variants

SPEKE [Jab96] was one of the earliest published protocols. The first security analysis of SPEKE has been given by MacKenzie within the BMP simulation based model [Mac01]. Later it has been shown that in fact multiple password guessing with one impersonation is actually feasible [Zha04].

SPEKE variants inherit the property from Diffie-Hellman key exchange, that a man-in-the-middle attacker has the possibility to modify both honest parties' resulting DH keys unless the whole transcript of the communication is used for generating the session key [HS14]. If the attacker replaces both intermediate exponentiations  $B^a$  and  $B^b$  of the honest participants by a fixed power  $(B^a)^c$  and  $(B^b)^c$  both parties will still obtain the same key. Note that this "attack" does not actually affect security in practice [HS14].

The original SPEKE protocol suffers from a weakness allowing for simultaneously testing more than one password per online connection [Zha04]. As a mitigation, hashing of the password was suggested in [Jab97], however without fully resolving the problem [LW15].

Unfortunately, the security proofs for SPEKE do not cover the case of elliptic curves. However, SPEKE-related constructions TBPEKE [PW17] and PACE [BFK09] have been proven secure, however those require more exponentiations. It was sometimes conjectured that SPEKE provides forward secrecy, but according to our knowledge this has not been formally proven.

One notable advantage of SPEKE and some SPEKE variants such as PACE [BFK09] when implemented on elliptic curves with integrated mapping [CGIP12] is that no full group operations are required, e.g., allowing also for so-called  $x$ -coordinate-only implementations. This feature provides some advantages of SPEKE and SPEKE variants in comparison with, e.g., SPAKE [AP05] and its variants and PAK[BMP00]. This way no full group operations are required for implementations but only a less-strict notion of a group modulo negation, that allows for scalar multiplication and differential addition but not for arbitrary point additions.

Verifier-based variants of SPEKE have been suggested already in 1997 [Jab97], unfortunately without a corresponding proof.

## 2.5 Review of the UC framework

In this paper we assume some familiarity with the framework of Universal Composability (UC). As a short introduction, we will give a summary of the essence here. For more details, we refer the reader to [Can00].

The general idea of UC is to define security in terms of idealized functionalities  $\mathcal{F}$  which provide services to a set of players  $P_i$ . Moreover the framework considers an adversary  $\mathcal{A}$  and an environment  $\mathcal{Z}$  and a real-world protocol  $\pi$  whose security is to be analyzed. In the context of UC all of the algorithmic strategy of  $\mathcal{A}$ ,  $\mathcal{Z}$  and  $\pi$  are provided in form of code for an interactive Turing machine (ITM). In an actual real-world execution, a plurality of interactive Turing machine instances (ITI) is generated upon request of the environment  $\mathcal{Z}$ . For instance several ITI may execute the ITM algorithm  $\pi$  for the parties  $P_i$ . Also the environment  $\mathcal{Z}$  and the adversary  $\mathcal{A}$  are given their respective ITI instance.

In the UC framework this "real-world" case is compared with an "ideal-world" case where the protocol  $\pi$  is replaced with the ideal functionality  $\mathcal{F}$  and the real-world adversary  $\mathcal{A}$  with an ideal-world adversary  $\mathcal{S}$ . The security model is based on the observation that if any (polynomially bounded) environment algorithm  $\mathcal{Z}$  cannot distinguish between the "real" and "ideal" world executions with any significant advantage, then using instances of protocol  $\pi$  is just as secure as using the ideal functionality  $\mathcal{F}$ .

From the perspective of the players  $P_i$ ,  $\mathcal{F}$  provides a set of subroutine calls that calculate a given function. For instance the subroutine call of the ideal functionality of a



PAKE protocol  $\mathcal{F}_{\text{pwKE}}$  returns a session key.

The definition of the algorithm of the ITM  $\mathcal{F}$  makes sure that sensitive information is hidden from the adversary as long as no "corruption" of parties occurs. Thus, the security targets are inherently guaranteed. With corruptions, we model the case that the adversary gets control over some of the protocol partners and, e.g., is able to retrieve data from that partner's internal memory. In the literature different types of corruption could be distinguished. In so-called "static corruptions" the adversary may gain control only just before starting with an actual protocol execution. In the so-called adaptive or Byzantine corruption models we give the attacker more power by allowing him to corrupt parties at any time during protocol execution. In this paper we consider the stronger Byzantine corruption.

The original UC theorem from [Can01] allows to analyze the security of a system viewed as a single unit, but it does not guarantee anything if different protocols share some amount of state and randomness, such as, e.g., a hash function functionality. For this reason for our application, the UC theorem cannot be used as-is. Our analysis, just as the strongly related work in [ACCP08] is thus implemented in the framework of universal composition with joint state [CR03].

## 2.6 Advantages and drawbacks of security proofs within the UC framework

When comparing the UC framework with alternative approaches, notably the game-based methodology a number of conceptual differences could be distinguished.

One of the most important advantages is that in the UC framework it is possible to follow a modular approach where more complex solutions could be composed of UC secure submodules by use of the composition theorem [Can00]. This avoids the need for detailed reduction-proofs of composed systems. This fact is strongly linked to the special role of the session id *sid* individually identifying one of possibly many copies of the ideal functionalities. Within the UC framework, the session id needs to be established prior to entering the protocol.

Note that depending on the actual publication, the session id generation is treated with more or less rigor. In this paper we consider this complexity and include the corresponding two additional messages for *sid* establishment in our protocol specification. In other work, this aspect seems to be treated with somewhat less rigor.

It is worth noting, that the integration of Diffie-Hellman in composable frameworks typically generates technical challenges [KR17], specifically when considering Byzantine adversaries. It is often infeasible for the simulator to provide the adversary secret scalars being consistent with the previously published group elements. Many previous security proofs were, thus, forced to restrict the analysis to the weaker static adversary model (e.g., [GMR06]). We consider it to be a somewhat exceptional fact that in our proof we were able to circumvent this challenge.

Finally it is worth noting that in the UC framework, the advantage of composability often comes at the expense of somewhat less quantitative formulas regarding computational complexity bounds. Commonly in the UC approach probabilities of algorithmic substeps are qualitatively assumed to be either negligible or not negligible. On this basis it is derived that the real-world protocol could be distinguished from the ideal functionality with negligible or non-negligible probability, often without giving explicit formulas regarding the exact bounds regarding the individual complexity assumptions. It is worth noting that there exist more elaborate composable frameworks considering this aspect with more rigor, see, e.g., [KR17]. This aspect, however, is out of the scope of this paper.

## 2.7 Overall proof strategy used in this paper

In this paper we try to avoid the introduction of new security notions and new ideal functionalities where possible. We first prove that the balanced sub-protocol CPace securely implements the ideal functionality  $\mathcal{F}_{\text{pwKE}}$  from [CHK<sup>+</sup>05] (repeated for reference in figure 6).

We then show by using the UC composition theorem that the combination of the sub-protocols securely implements  $\mathcal{F}_{\text{apwKE}}$  from [GMR06] and thus provides conventional "full" augmentation.

We then show that when the ephemeral Diffie-Hellman step within AuCPace is replaced by a long-term key variant, the computational complexity for the server is essentially halved, while preserving the most important guarantee. We will present the corresponding idealized functionality  $\mathcal{F}_{\text{papwKE}}$  together with the security proof for partially augmented AuCPace.

## 3 The AuCPace protocol

### 3.1 Design rationales for the AuCPace protocol

Our dominating guideline for the protocol development was server-side efficiency. In a typical use case of a remote human-machine interface (HMI) for a resource-constrained target, the HMI client-side has much more powerful computing capabilities.

We assessed power consumption to be one major issue in line with the results of [HL17]. Note that this goes beyond just minimizing the number of exponentiations. Also the choice of a suitable curve, and the need for point compression and point verification sub-steps should be considered. We observed, e.g., that  $x$ -coordinate-only Diffie-Hellman implementations could provide significant advantages, regarding all of speed, ease-of-implementation and memory-consumption. We therefore searched for constructions not requiring full group operations, because this could inherently avoid the need for point compression and verification. Specifically on curves with twist security this could allow for more efficient methods against small-subgroup[LL97] attacks.

We also tried to minimize the number of required primitives in the protocols. E.g. we aimed at avoiding the digital signature scheme needed for some constructions, such as the  $\Omega$ -Method from Gentry, MacKenzie and Ramzan [GMR06].

For AuCPace, we also aimed at aggressively minimizing memory requirements. E.g., it was considered highly desirable, to try to avoid storage for full communication transcripts. Actually as a side-effect of the proof strategy used for AuCPace we don't execute many protocol substeps in parallel and, thus don't require large memory buffers for temporaries and state. Throughout the calculation, we aimed also at minimizing the memory requirement for temporary variables. E.g. for calculating scalar multiplications on elliptic curves we searched for efficient strategies not requiring large pre-computed tables, as e.g. required for window-based speedup techniques. We assumed also that often the ROM limitation might not allow for the tables typically needed for fixed-base point speedups.

### 3.2 Parameters of the AuCPace protocol

The AuCPace protocol is depicted in figures 3 and 4. The protocol is parametrized by

- A password based key derivation function  $\text{PBKDF}_\sigma$  that is itself parametrized by algorithm settings  $\sigma$ , specifying e.g. the memory consumption for the password hash or an iteration count.  $\text{PBKDF}_\sigma$  calculates a string from the password  $pw$ , a username and a so-called "salt" value. For our reference implementation of AuCPace, we use the memory-hard scrypt [PJ12] password hash parametrized for a memory consumption of 32 MByte.

- A (hyper-)elliptic curve  $\mathcal{C}$  with a group  $\mathcal{J}$  with co-factor  $c_{\mathcal{J}}$  and a Diffie-Hellman (DH) protocol operating on both,  $\mathcal{C}$  and its quadratic twist  $\mathcal{C}'$ . We denote the DH base point in  $\mathcal{J}$  with  $B$ . We don't require full group structure in  $\mathcal{J}$  but could also instantiate AuCPace if only group operations modulo negation are available. This can result in more efficient implementations. (For more details regarding this efficiency aspect see the corresponding discussion for the qDSA/EdDSA signature schemes in [RS17].). For our reference implementation, we use Curve25519 [Ber06] and the  $x$ -coordinate-only Diffie-Hellman protocol X25519. In this paper we follow the recommendation in [Ber14] and reserve the name Curve25519 for the curve and X25519 for the protocol. For the DH protocol we mostly use a simple exponentiation notation, even if additional co-factor handling and clamping might apply for the scalars for guaranteeing that the result of any exponentiation is always in  $\mathcal{J}$ . (The exception from this rule is that we decided explicitly detail the co-factor  $c_{\mathcal{J}}$  complexity in figures 3 and 4 because this handling is actually crucial for security.)
- An encoding that represents either a point  $Y$  on  $\mathcal{J}$  or on the quadratic twist in a fixed-size bit stream. In our reference implementation we make use of an encoding of the  $x$ -coordinate of the point on Curve25519.
- A verification algorithm that checks whether the order of an encoded element  $Y$  within  $\mathcal{C}$  or  $\mathcal{C}'$  is large enough for the security target specified by the complexity of the computational Diffie-Hellman problem (CDH) for security parameter  $k$ . In our reference implementation we make use of Curve25519's twist security and the integrated co-factor of 8 for X25519 scalars, i.e., we just verify that  $X25519(x, Y) \neq 0$ .
- A Map2Point operation and its inverse map  $\text{Map2Point}^{-1}$ .  $\text{Map2Point}(s)$  is required to map a string  $s$  to a point from a cryptographically large subgroup  $\mathcal{J}_m$  of  $\mathcal{C}$ , such that the discrete logarithm of the point is unknown. The inverse map  $s = \text{Map2Point}^{-1}(X, l)$  is required to map a point  $X \in \mathcal{J}_m$  to a bit string  $s$  of length  $l$  bits such that for any randomly sampled  $X \in \mathcal{J}_m$  the string  $s$  is indistinguishable from a random bit string of length  $l$ . For our reference implementation we use Elligator2 introduced by Bernstein, Hamburg, Krasnova and Lange in [BHKL13] on Curve25519, where the sign of the inverse map result is chosen at random.
- Hash functions  $H_0 \dots H_4$ . For our reference implementation we use SHA512 where the hash function index is prepended as little-endian four-byte word. (We chose SHA512 for having a high security margin. Imperfections of the hash function for proofs using the random-oracle model, should be considered to be particularly critical.)

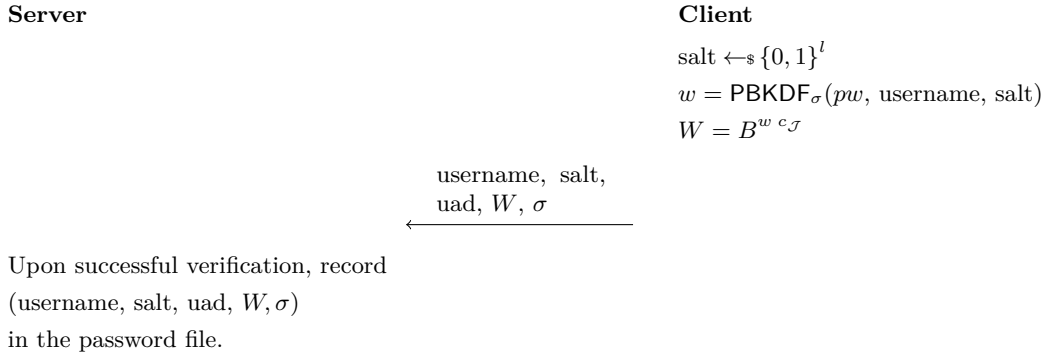
We will refer to our reference implementation of AuCPace using the actual choices above as AuCPace25519. While denying any legal responsibility, the authors declare that they are not aware of any intellectual property right or patent limiting the use of AuCPace25519.

### 3.3 Configuring the password verifier on the server

Two basic sub-protocols could be distinguished. In a first sub-protocol the server is given a password-verifier  $W = B^w$  for storage in its database. This protocol is depicted in figure 3. The second sub-protocol, shown in figure 4, uses the available password verifier for establishing a session key.

The configuration of password verifiers requires one message. We assume, that the specific group  $\mathcal{J}$  and the permissible set of PBKDF parametrizations  $\sigma$  of the server are known to the client. The client chooses a fresh "salt" value and hashes the password  $pw$  to yield a secret scalar  $w$ . Then a password verifier  $W = B^w$  is calculated and sent to the server for storage in the database. Optionally user authorization data (uad) is also

## Store password operation for AuCPace



**Figure 3:** Protocol for password configuration. The optional data element uad represents application data associated with this specific user account, e.g. specifying the granted authorization level on the server.

transmitted. The server then checks whether the parametrization  $\sigma$  and the authorization setting to attribute to the user are acceptable and stores the verifier  $W$  in the database together with the salt and the authorization settings.

Note that all of the protocols in this paper choose ephemeral fresh "salt" values upon password changes. In some settings, though, we see reasons for using a fixed "pepper" that is chosen once and for all users and servers of a specific plant. When using an ephemeral salt value, we could not avoid some leakage of information. Specifically, with ephemeral "salts" it could not completely be hidden whether or not a user entry is available in the server's database and that a user has changed his password. On the other hand, using a fixed "pepper" known to many parties has the disadvantage that an attacker can pre-compute so-called rainbow-tables.

In line with other V-PAKE papers ([PW17, CHK<sup>+</sup>05, JKX18]) for the scope of this paper we concentrate on the security proof for the session establishment only, i.e., for this first sub-protocol we assume that communication is using a confidential channel. We also assume that the client is properly authenticated and has sufficient privileges. More formally, we do not consider adversaries  $\mathcal{A}$  that read or modify messages of this sub-protocol.

It is important to note that the password is *never* passed in clear-text to the server. This also implies that the computationally complex PBKDF function is calculated on the client. Note also that by letting the client choose the "salt" value, we provide a path for distributing password verifiers from a centralized user credential server by use of an offline ticket mechanism, i.e., using a single unidirectional message.

With respect to the generation of the password verifier  $W$  our protocol shares some similarities with AugPake [SKI10] and VTBPEKE [PW17] which also use a group element  $W = B^{\xi}$  as verifier where a password-derived key  $\xi$  is used as a secret exponent.

### 3.4 Establishing session keys based on the password $pw$ and the password verifier $W$

Establishment of the session key  $sk$  is realized by a sequence of several steps shown in figure 4. First a sub-session id  $ssid$  is established as required by the UC framework prior to entering the protocol. (An alternative option would have been to obtain  $ssid$  just by concatenating the two random nonces  $s$  and  $t$ . [BLR04])

Secondly, a password related string  $PRS$  is calculated. We refer to this sub-protocol as the AuCPace augmentation layer. Establishing  $PRS$  involves one message round. After learning the user name, the server fetches the "salt" value from the database. In case that

## AuCPace

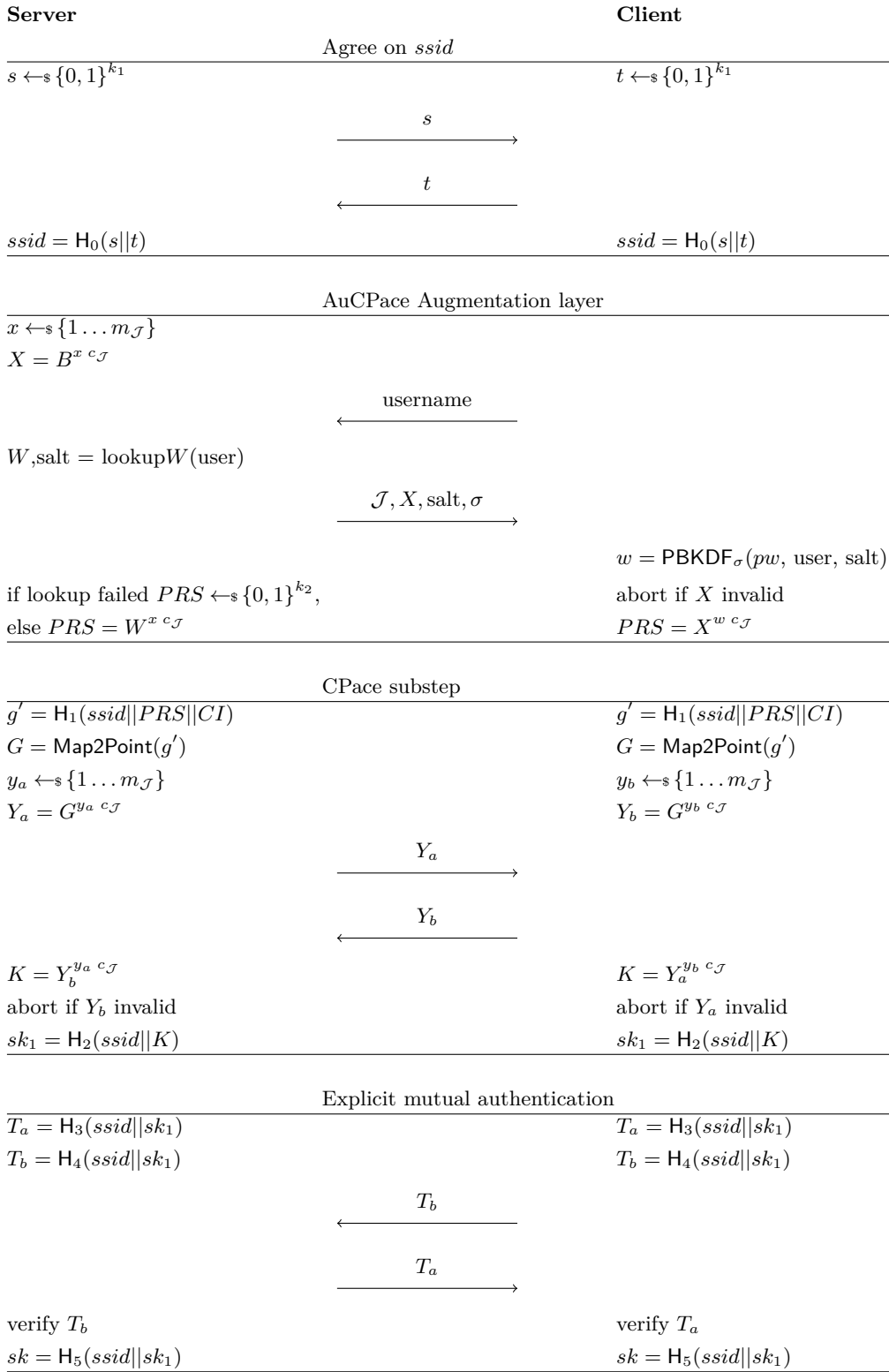


Figure 4: Protocol AuCPace.

no entry is available for the user name, we suggest to hash a server-specific secret constant together with the username in order to yield a dummy salt value.

The server then transmits its "salt" string together with the value  $\sigma$  required for parameterizing the password hash  $\text{PBKDF}_\sigma$  and an ephemeral public key  $P = B^{p^{c\mathcal{J}}}$  and the group representation  $\mathcal{J}$  to use for the Diffie-Hellman protocol. Both, server and client entity then calculate a password-derived string  $PRS$ . While the server uses the password verifier  $W$  from its database, the client has to calculate  $\text{PBKDF}_\sigma$ . We neither expect to have permanent storage available on the client, nor do we recommend any permanent caching of "w". If no entry is available for  $W$  in the server's password file, or if the point verification on the client fails, the protocol is continued with a randomly sampled  $PRS$  string instead of aborting. Our approach somewhat mitigates the fact that the openly communicated "salt" value leaks some information on the server's password file contents. (At the same time we have to accept more workload when facing some types of denial-of-service attacks.)

Then client and server enter the balanced sub-protocol CPace with the password-derived string  $PRS$  as password. There, first an ephemeral generator  $G$  is calculated by use of the Map2Point algorithm.

Calculation of  $G$  involves a "channel identifier" CI which is hashed together with the  $PRS$ . In the context of TCP/IP, the CI might be constructed by concatenating unique representations of the server's and client's IP address and TCP port numbers. Hashing CI into  $G$  allows us to fend off certain types of relay attacks.

Incorporating the *ssid* into the calculation guarantees the generator  $G$  to be ephemeral also in the partially augmented setting from section 6.1. Note that prefixing the session id is also suggested in section 10.3 of [KTR13] for a security analysis in the IITM model [KTR13] in case that one global random oracle is to be used for an unlimited numbers of subsessions.

After determining  $G$  the two parties implement a Diffie-Hellman protocol by exchanging group elements  $Y_a$  and  $Y_b$  and deriving a shared secret point  $K$ . Note that it is mandatory for the receiving party to verify the points  $Y_a$  and  $Y_b$ . Then a first session key  $sk_1$  is derived from  $K$ .

As last sub-protocol, optionally explicit authentication is added by exchange of two authenticator messages  $T_a$  and  $T_b$ . Finally the session key  $sk_1$  is refreshed to yield the final session key  $sk$ .

With respect to the mandatory point verification, we do not impose the conventional requirement that the implementation has to verify that the points  $X$ ,  $Y_a$  and  $Y_b$  are  $\in \mathcal{J}$ . Instead we impose a less strict requirement that could be implemented more efficiently on some curves, notably if they have secure twists: We require the verification that the order of the respective points is large with respect the required complexity assumption for the CDH problem, such as to prevent collisions in the resulting points  $K$  and  $X^w$  due to small subgroup attacks.

### 3.5 Key difference between AuCPace and previously known SPEKE-based constructions

The main difference between SPEKE and the balanced CPace subprotocol shown here lies in the fact that CPace works with an ephemeral generator that depends on the session id and the channel identifier CI. This allows for both, a natural way for proving security in the UC framework and a memory-optimized approach for implicitly authenticating data such as the CI.

The feature of the ephemeral generator is shared between CPace and PACE. Unlike PACE, CPace does not need an additional block cipher most certainly employed for patent circumvention only. Note that this reduces both, implementation complexity and



The protocol CPace is parametrized by a security parameter  $k$ . CPace implements the hashes with output size  $2k$   $H_1$  and  $H_2$  by the functionality  $\mathcal{F}_{\text{RO}}$ . The protocol also uses a Diffie-Hellman key exchange protocol (written in exponentiation notation  $X^y$ ) operating on points  $X$  on a group modulo negation  $\mathcal{J}$  of order  $m_{\mathcal{J}}$  and its twist. It also uses a `Map2Point` primitive mapping a string of length  $2k$  on an element  $Y$  where  $Y^r$  generates  $\mathcal{J}$ . It finally uses a `Verify`( $Y$ ) algorithm checking for the subgroup order of a given  $Y$  on either  $\mathcal{J}$  or the twist.

**Protocol:**

1. When  $P_j$  receives input (`NewSession`,  $sid, P_i, P_j, pw$ , responder role), he calculates  $g' = H_1(sid || P_i || P_j || pw)$  and  $G = \text{Map2Point}(g')$ . He then samples a fresh nonzero scalar  $y_b$  with  $0 < y_b < m_{\mathcal{J}}$  and calculates  $Y_b = G^{y_b}$ . He sets up a session record ( $sid, P_i, y_b, Y_b$ , responder role) and marks it as `fresh`. He then waits for a (`CPace initiator`,  $sid, P_i, Y_a$ ) message.
2. When  $P_i$  receives input (`NewSession`,  $sid, P_i, P_j, pw$ , initiator role), he calculates  $g' = H_1(sid || P_i || P_j || pw)$  and  $G = \text{Map2Point}(g')$ . He then samples a fresh nonzero scalar  $y_a$  with  $0 < y_a < m_{\mathcal{J}}$  and calculates  $Y_a = G^{y_a}$ . He sets up a session record ( $sid, P_j, y_a$ , initiator role) and marks it as `fresh`. He then sends (`CPace initiator`,  $sid, P_i, Y_a$ ) to  $P_j$ .
3. When  $P_j$  receives input (`CPace initiator`,  $sid, P_i, Y_a$ ) and finds a `fresh` session record ( $sid, P_i, y_b, Y_b$ , responder role) it sends message (`CPace responder`,  $sid, P_j, Y_b$ ) message to  $P_i$ . He then calculates `Verify`( $Y_a$ ). If the point order is sufficient for security parameter  $2k$  it calculates  $K = Y_a^{y_b}$  and  $sk = H_2(sid || K)$  and outputs ( $sid, sk$ ). It aborts otherwise. In either case it marks the session record as `completed`.
4. When  $P_i$  receives input (`CPace responder`,  $sid, P_j, Y_b$ ) and finds a `fresh` session record ( $sid, P_j, y_a$ , initiator role) he calculates `Verify`( $Y_b$ ). If the point order is sufficient for security parameter  $2k$  it calculates  $K = Y_b^{y_a}$  and  $sk = H_2(sid || K)$  and outputs ( $sid, sk$ ). It aborts otherwise. In either case it marks the session record as `completed`.

**Figure 5:** CPace Protocol definition for the proof of indistinguishability.

the number of assumptions required for the proof. For CPace there is no need for the ideal-cipher assumption.

Most importantly, AuCPace allows for two different levels of augmentation. If an ephemeral key pair  $(x, X)$  is used, we obtain the conventional security guarantees of an augmented PAKE protocol. Additionally, AuCPace allows for a "partially augmented" mode where the key pair  $(x, X)$  is of long-term type and generated once during password registration. The security guarantee of partial PAKE augmentation will be formally introduced in section 6. For this reason we will continue the more detailed comparison between AuCPace and other protocols only in section 7. We invite readers wishing to directly continue with this discussion to skip the rather technical proof sections 4 to 6 for a first reading.

## 4 Proof of indistinguishability for the balanced sub-protocol CPace

In this section we will deal with the balanced PAKE protocol CPace corresponding to the middle part of figure 4. We formalize this sub-protocol in figure 5.

In this proof we show that the protocol in figure 5 emulates the functionality  $\mathcal{F}_{\text{pwKE}}$  (figure 6) from [CHK<sup>+</sup>05].  $\mathcal{F}_{\text{pwKE}}$  receives the passwords  $pw$  from the environment-controlled parties  $P_i$  and  $P_j$  and returns upon a `NewKey` request the same random session key  $sk$  if and only if (iff) the passwords match.

For the security analysis in this section we need to map the inputs and outputs of the

protocol from figure 4 to the notation used for defining the ideal functionality  $\mathcal{F}_{\text{pwKE}}$  in figure 6. The password related components ( $W^x$  and  $X^w$  respectively) correspond to the passwords  $pw$  and the resulting session key  $sk$  from figure 6 corresponds to  $sk1$  in figure 4. We define the channel identifier  $CI$  to be the concatenation of  $P_i$  and  $P_j$ .

## 4.1 Proof strategy

Our proof closely follows the strategy from [ACCP08]. We also use a sequence of games  $G_0$  to  $G_4$  in which the simulator algorithms  $\mathcal{S}_0$  to  $\mathcal{S}_4$  are executed. We organize these algorithms  $\mathcal{S}_n$  as a combination of independent ITI that only interact through their well-defined APIs and have internal state (tape) that is not accessible from the outside. Initially we have one such ITI for each simulated honest party  $P_i$ , the hash functionality  $\mathcal{F}_{\text{RO}}$  and one ITI executing the algorithm of the real-world adversary  $\mathcal{A}$ . We let  $\mathcal{S}_n$  invoke the other ITI during the course of the execution, i.e., we treat the real-world adversary algorithm  $\mathcal{A}$  as a black box subroutine library for  $\mathcal{S}_n$ .

When  $\mathcal{A}$  decides to corrupt a party  $P_i$ , we need to provide it all of the corrupted ITI's internal accessible state. The subsequent behavior of this party is then controlled by  $\mathcal{A}$ . The adversary is also given the secret scalars used in the real-world protocol. Finally, we give  $\mathcal{S}_n$  also access to an ITM  $\mathcal{F}_n$  in each game, where  $\mathcal{F}_0$  is initially not providing any service. In each game we extend the functionality of  $\mathcal{F}_n$  until it implements  $\mathcal{F}_{\text{pwKE}}$ .

In the games in our proof we re-factor the algorithms  $\mathcal{S}_n$  such that each change is indistinguishable for the environment  $\mathcal{Z}$ .

At the end of the game sequence, we end up with an algorithm  $\mathcal{S}_4$  that makes only calls to  $\mathcal{F}_4$  which itself implements exactly the ideal functionality  $\mathcal{F}_{\text{pwKE}}$ .

## 4.2 Game-based proof

**Theorem 1.** *The protocol CPace from figure 5 securely realizes  $\mathcal{F}_{\text{pwKE}}$  in the  $\mathcal{F}_{\text{RO}}$  hybrid model in the presence of adaptive adversaries under the assumptions*

- of the hardness of the computational Diffie-Hellman problem in  $\mathcal{J}$  (A0)
- that the number of elements  $X \in \mathcal{J}_m$  is not significantly smaller than the order of the group and that for any given randomly sampled group element  $X \in \mathcal{J}_m$  the inverse map  $\text{Map2Point}^{-1}(X, l)$  is indistinguishable from a random string of length  $l$  (A1)
- that for any base point  $B$  and uniformly distributed random string  $s$ ,  $\text{Map2Point}(s)$  is close to uniformly distributed within the image of  $\text{Map2Point}$  and the probability for finding an exponent  $y$  such that  $B^y = \text{Map2Point}(s)$  is negligible. (A2).

Note that the assumption A1 is covered by the security guarantees of maps such as Elligator2 [BHKL13] and will allow us for covering the case of fully adaptive adversaries. Without this assumption we would not be able to resolve the commitment problem linked to the Diffie-Hellman substep for adaptive adversaries.

Assumption A2 closely matches the requirements from [BFK09] for the  $\text{Map2Point}$  operation in PACE.

### 4.2.1 Game $G_0$ : Real Game

$G_0$  is the real game in the random-oracle model using the functionality  $\mathcal{F}_{\text{RO}}$  from figure 7. The parties  $P_i$  receive `NewSession` queries from all simulated honest parties. These queries contain the passwords provided by the environment  $\mathcal{Z}$ .  $P_i$  then executes the actions of initially honest parties in the protocol. In the event of corruptions, the internal state of the parties is passed to the real-world adversary algorithm  $\mathcal{A}$ . The subroutine library  $\mathcal{F}_0$  is empty.

The functionality  $\mathcal{F}_{\text{pwKE}}$  is parametrized by a security parameter  $k$ . It interacts with an adversary  $\mathcal{S}$  and a set of parties via the following queries:

**Upon receiving a query** ( $\text{NewSession}, sid, P_i, P_j, pw, \text{role}$ ) **from party**  $P_i$  :

Send ( $\text{NewSession}, sid, P_i, P_j$ ) to  $\mathcal{S}$ . In addition, if this is the first  $\text{NewSession}$  query, or if this is the second  $\text{NewSession}$  query and there is a record  $(P_j, P_i, pw')$ , then record  $(P_i, P_j, pw)$  and mark this record fresh.

**Upon receiving a query** ( $\text{TestPwd}, sid, P_i, pw'$ ) **from the adversary**  $\mathcal{S}$  :

If there is a record of the form  $(P_i, P_j, pw)$  which is fresh, then do: If  $pw = pw'$ , mark the record compromised and reply to  $\mathcal{S}$  with "correct guess". If  $pw \neq pw'$ , mark the record interrupted and reply with "wrong guess".

If there is a record of the form  $(P_i, P_j, pw)$  which is completed with session key  $sk$

and there is no record of the form  $(P_j, P_i, pw^*)$  which is completed

and this is the first  $\text{TestPwd}$  query for  $sid$  then do:

If  $pw = pw'$ , reply to  $\mathcal{S}$  with  $sk$ . If  $pw \neq pw'$ , reply with "wrong guess".

**Upon receiving a query** ( $\text{NewKey}, sid, P_i, sk$ ) **from**  $\mathcal{S}$  **where**  $|sk| = k$ :

If there is a record of the form  $(P_i, P_j, pw)$ , and this is the first  $\text{NewKey}$  query for  $P_i$ , then:

- If this record is compromised, or either  $P_i$  or  $P_j$  is corrupted, then output  $(sid, sk)$  to player  $P_i$ .
- If this record is fresh, and there is a record  $(P_j, P_i, pw')$  with  $pw' = pw$ , and a key  $sk'$  was sent to  $P_j$  and  $(P_j, P_i, pw)$  was fresh at the time, then output  $(sid, sk')$  to  $P_i$ .
- In any other case, pick a new random key  $sk'$  of length  $k$  and send  $(sid, sk')$  to  $P_i$ .

Either way, mark the record  $(P_i, P_j, pw)$  as completed.

**Figure 6:** Ideal functionality  $\mathcal{F}_{\text{pwKE}}$  re-presenting balanced PAKE without explicit authentication. The text without the grey section exactly corresponds to the definition from [CHK<sup>+</sup>05]. Note that in our proof we need to extend this functionality by allowing for one single  $\text{TestPwd}$  query also if a session key has already be issued to the honest party as detailed in the highlighted grey extension text. We refer to this extended functionality as "relaxed"  $\mathcal{F}_{\text{pwKE}}$  (in line with the naming of a corresponding extension that was introduced in the context of the proof for the protocol OPAQUE in section 5.4 of [JKX18]).

#### 4.2.2 Game $G_1$ : Simulation of the random oracle

Here we modify the previous game by replacing the calls  $H_n(q)$  to the original  $\mathcal{F}_{RO}$  hash ITI by a separate implementation. We let  $\mathcal{S}_1$  maintain an initially empty list  $\Lambda$  of value pairs  $(n, q, g, r)$ . For any hash query  $H_n(q)$  such that  $(n, q, *, r)$  appears in  $\Lambda$  from any of the ITI libraries, the returned answer is  $r$ . In case that no query  $q$  has yet occurred, we handle separately the cases of  $n = 1$  and  $n \neq 1$ . In case of  $n \neq 1$  we implement the conventional random-oracle model by choosing a new random  $r$  of length  $k$ , by storing  $(n, q, 0, r)$  in  $\Lambda$  and by returning  $r$  to the calling ITI.

For  $n = 1$  instead, we aim at generating a random string  $r$  such that the discrete logarithm of the point  $\text{Map2Point}(r)$  is known. For this purpose we first generate a random point  $G$  whose discrete logarithm is known and use the inverse map  $\text{Map2Point}^{-1}(G, k)$  for converting it into a bit string of length  $k$ . We use the guarantee, that for any random point  $G$  the string  $r = \text{Map2Point}^{-1}(G, k)$  is indistinguishable from a random value.

For calculating the random point  $G$ , we first choose a random nonzero value  $g$  being smaller than the order of the group. We calculate the point on  $\mathcal{J}, G = B^g$ . We then test whether  $G$  is in the image of  $\text{Map2Point}$ ,  $\mathcal{J}_m$ . If  $G$  is not in the image, we restart with a new random value  $g$  until  $G \in \mathcal{J}_m$ . This is guaranteed to succeed in probabilistic polynomial time because  $\mathcal{J}_m$  is large by assumption A2.

Then we calculate  $r = \text{Map2Point}^{-1}(G, k)$ , record  $(1, q, g, r)$  in  $\Lambda$  and return  $r$ .

Since the inverse map returns a string indistinguishable from a random string by the assumption A1  $G_0$  and  $G_1$  are indistinguishable for the environment.

#### 4.2.3 Considering the chance that an adversary guesses $sk$ by chance. (former paper versions: $G_2$ )

In this step, we consider the probability of obtaining the session key  $sk$  without querying the hash oracle  $H_2$  for  $K$ . We assume that the probability of successfully guessing  $sk$  is the same as the probability of guessing  $K$ . (In earlier versions of this proof we did want to make the simulator to follow different strategies, however it turned out that we could not always detect this when facing the UC dummy adversary and do not actually need the adaptations of making the simulator abort.  $G_2$  was retained in the proof in order to keep the numbering consistent with the earlier versions.  $G_2$  is now un-modified in comparison to  $G_1$  and thus indistinguishable. ).

#### 4.2.4 Game $G_3$ : Restrict the access to the password.

Here, when receiving the passwords for party  $P_i$  from the environment, we let  $P_i$  pass them directly to the subroutine library  $\mathcal{F}_3$  and allow the rest of the program  $\mathcal{S}_3$  no longer access the password unless the simulated party  $P_i$  gets corrupted. We let  $\mathcal{S}_3$  inform  $\mathcal{F}_3$  in case that a party got corrupted such that  $\mathcal{F}_3$  returns the password in this case. We add also an implementation of the  $\text{TestPw}$  query to  $\mathcal{F}_3$  and implement it according to the spec. of  $\mathcal{F}_{pwKE}$ . In this game, we preliminarily also add a  $\text{SamePw}$  query to  $\mathcal{F}_3$  that returns true if the passwords match.

As a result, the password-derived generator  $G$  from figure 4 is no longer known to the ITI  $P_i$ . We therefore cannot start the original protocol from the beginning and need to refactor the ITM for the parties as well.

Instead we let the ITI  $P_i$  calculate the protocol messages  $Y_a$  and  $Y_b$  to be random multiples of the group's base point  $B$ ,  $Y_a = B^{z_a}$  and  $Y_b = B^{z_b}$ . We also let  $P_i$  maintain an initially empty list  $\Gamma$  and store the secret scalars of simulated honest parties  $(sid, P_i, y \text{ unknown}, z_a)$  and  $(sid, P_j, y \text{ unknown}, z_b)$  in this list together with the respective session id.

Since powers of  $G$  from game  $G_2$  generate  $\mathcal{J}$ , the resulting points  $Y_a, Y_b$  take any value on  $\mathcal{J}$  with equal probability (except for the neutral element) for honest parties. So do

the points generated in  $G_3$ . The public messages in  $G_2$  and  $G_3$  are, thus, indistinguishable for the environment.

As soon as both messages  $Y_a$  and  $Y_b$  have been delivered by  $\mathcal{A}$  we calculate the Diffie-Hellman results using the received points and the local exponents  $z_a$  and  $z_b$ . If the respective result points differ, we know that  $\mathcal{A}$  has modified the messages in a destructive way and record for this session a flag `DHFAILs`.

In case that a party  $P_i$  gets corrupted before calculating  $K$ , we need to hand over  $\mathcal{A}$  something being consistent with the internal state from  $P_i$  in  $G_2$ , notably the values  $y$  ( $y_a$  or  $y_b$  respectively for client and server).

In the case of corruption,  $\mathcal{F}_3$  grants us access to the secretly stored  $pw$  from its internal state.  $\mathcal{S}_3$  may then take the password and the session id and make a corresponding hash query to  $H_1$ . We then retrieve the secret scalar value  $g$  from  $\Lambda$ . We fetch the party  $P_i$ 's secret scalar  $z$  from  $\Gamma$  and calculate  $y = z/g$ . We add the party's secret scalar  $z$  to the record in  $\Gamma$  with  $y$  and hand over  $y, pw$  to  $\mathcal{A}$ .

In case that any party gets corrupted after calculating  $K$  but before calculating the final  $H_2$ , we perform the secret scalar correction above and recalculate a new  $K = Y_r^y$  by using the received point value  $Y_r$  and pass  $K$  to  $\mathcal{A}$ .

The code for the verification handling for the received points  $Y_r$  can remain unchanged in comparison to Game  $G_2$ .

In case that the point verification fails for any party, we do not generate a session key and do not need to calculate the final hash  $H_2(sid||K)$ . In case that the final hash  $H_2$  needs to be calculated for the first of the parties  $P_i$  and  $P_i$  is still honest, we need to provide a session key to  $P_i$ . (Note that this could be either server or client.) We distinguish three cases.

- If the other party was corrupted earlier, we know the other party's password  $pw'$ . We then may issue a `TestPw` query to  $\mathcal{F}_3$ . If the guess was correct, we learn the local secret scalar value  $y$  by the method described above and calculate  $K = Y_r^y$  with the received remote point  $Y_r$ . We query  $sk = H_2(sid||K)$  for the corrected value of  $K$  and return the result to  $P_i$ . If the guess was wrong, we sample a new random key  $sk$  and return it to  $P_i$ .
- If the other party is still honest, we sample a new random key  $sk$  and send it to  $P_i$  and record this session key together with the session id and the party identifiers  $(sid, P_i, P_j, sk)$ .
- If the other party is impersonated by  $\mathcal{A}$  we also sample a new random key  $sk$  and send it to  $P_i$  and record this session key together with the session id and the party identifiers  $(sid, P_i, P_j, sk)$ . Note that (according to the previous game), we will be returning a distinguishable key  $sk$  iff  $\mathcal{A}$  somehow managed to guess the value  $K = Y_r^y$  in  $G_2$ . We will calculate the corresponding probability `GuessK` in section 4.4.

The remaining task is to calculate the session key  $sk$  for a second party  $P_j$  if it is not corrupted until the very end or corrupted before calculating  $sk$ . In any of these two cases, we know that two messages  $Y_a$  and  $Y_b$  must have been delivered by  $\mathcal{A}$ , and we, thus, have access to the `DHFAILs` marker and that the received points are not from a low order sub-group (or the neutral element). Also, because we know that we have to simulate session key generation for the second time, we know that the first party was honest until the end of the protocol.

If the second party is also honest until the very end, we make a `SamePw` query introduced temporarily to  $\mathcal{F}_3$ . If the passwords match and if the session is not marked as `DHFAILs`, we return the same  $sk$  value to  $P_j$  as for the first party, otherwise we sample a new random key  $sk'$  and return this one to  $P_j$ .

In case that the second party  $P_j$  gets corrupted after calculating  $K$  we first correct  $K$  using the secret exponent  $g$  retrieved from  $\Lambda$ . In case that we recognized destructive

modification of the Diffie-Hellman points by the `DHFAILS` marker for the session, we just sample a new value for  $sk'$  by the interface of the random oracle  $sk' = H_2(sid||K)$  and pass  $sk'$  to  $\mathcal{A}$ . There is only a negligible chance of collision with the key  $sk$  sent to the first party, since both  $sk$  and  $sk'$  have been randomly sampled. There is also only a negligible chance that  $\mathcal{A}$  managed to make both parties issue the same session key despite different passwords by modification of the transmitted points. For this reason  $G_3$  and  $G_2$  are indistinguishable for this case.

If the Diffie-Hellman points have not been modified in a destructive way (`DHFAILS` not recognized), the session key issued in  $G_2$  depends on the password. We learn the party's password  $pw$  from  $\mathcal{F}_3$ . We then may issue a `TestPw` query to  $\mathcal{F}_3$ . If the guess was correct, we have to provide the same session key to  $\mathcal{A}$  as for the first party if `DHFAILS` is not recognized. For this purpose, we program the value  $H_2(sid||K) := sk$  to the session key returned to the first party. This could fail only, if the oracle  $H_2$  already has been queried for  $(sid||K)$ , again corresponding to the probability `GuessK` that we deal with in section 4.4. (Note that it is for this re-programming operation that we will later need to be granted access to the session key issued to the client by the ideal functionality  $\mathcal{F}_{pwKE}$ . Otherwise we could not give  $\mathcal{S}$  access to the session key that would have been calculated by honest parties for corruptions occurring just after executing the hash function.)

The messages  $Y_a, Y_b$  generated in Game  $G_2$  and  $G_3$  are indistinguishable for the environment because they come from the same distribution. (Note that for this precise aspect, the appropriate co-factor handling was mandatory!) Also the session keys are sampled from an indistinguishable uniform distribution in both cases. Session keys delivered to parties  $P_i$  and  $P_j$  match under the same conditions as in  $G_2$ . Inserting points on the group's twist by the adversary always leads to different session keys for both parties, just as in  $G_2$ .  $G_2$  and  $G_3$  are, thus, indistinguishable for  $\mathcal{Z}$ .

In  $G_2$  the adversary is able to make an attempt to impersonate a honest party with a password  $pw'$ . I.e. the adversary follows the protocol just as a honest party. If the passwords match, the adversary will be calculating the same session key as the honest party. We need to simulate this situation also in  $G_3$  in a way that is indistinguishable for the environment. The behaviour is symmetric for initiator and responder, so we assume here that the adversary takes over the responder role.

We proceed as follows. We simulate the first message  $Y_a$  as  $Y_a = B^{y_a}$ , as discussed above. We will observe that the adversary queries the random oracle  $H_1$  for the specific  $sid$ , party identifiers and the password  $pw'$ . The simulator observes these queries and sets up a list  $G_l$  of candidate generators for this session id. Since game  $G_1$  the simulator knows the corresponding secret exponents  $g_l$  such that  $G_l = B^{g_l}$ . The adversary follows the protocol specification and calculates  $Y_b = G^{y_b}$  which we learn from the communication. Subsequently the adversary will query the random oracle  $H_2(sid||K)$  for the point  $K = Y_a^{y_b} = B^{y_a \times y_b}$ . The simulator does observe this query to  $H_2$ . In case that  $K$  matches  $Y_b^{y_a \times (1/g_l)}$  for one of the candidates, we learn the password  $pw'$ . We then use the `TestPw` query for finding out whether the passwords actually match. Depending on the result we provide the same or different session keys. We have control over the session keys passed to the honest party by  $\mathcal{F}_{pwKE}$  and the ability to program a chosen session key for  $H_2$ . Note that in order to allow for this simulation strategy, we need to be able to call the `TestPw` query also in case that a key already has been delivered to the honest party. This requires the extension of the relaxed version of  $\mathcal{F}_{pwKE}$  allowing for one single `TestPw` query also for a session that is already completed. In both games, both parties will be given matching session keys if the honest party and the adversary used the same password. If they did not use the same password, they will obtain different session keys with overwhelming probability in both games.



#### 4.2.5 Game $G_4$ : Merge the key generation procedures to the functionality $\mathcal{F}_4$ .

In this game we essentially only do code-refactoring and move the code responsible for session key generation to the ITM  $\mathcal{F}_4$  . We make  $\mathcal{F}_4$  implement exactly the functionality  $\mathcal{F}_{\text{pwKE}}$ . (Note that in line with conventions in UC  $\mathcal{F}_4$  gives access to the passwords  $pw$  in case of corruptions.) We remove the `SamePwd` query from the list of queries for  $\mathcal{F}_4$  because now,  $\mathcal{F}_4$  could easily check itself for password identity in its internal storage. Within  $S_4$  we finally replace the sampling of the session keys by calls to the `NewKey` query of the ideal functionality.

Since between  $G_3$  and  $G_4$  no functionality change is present,  $G_3$  and  $G_4$  are indistinguishable for  $\mathcal{Z}$ .

### 4.3 Dealing with the case that the adversary did tamper $Y_b$ and $Y_a$ in a non-destructive way

The conventional approach for cryptographic key establishment protocol design would be to hash all the transcript of the protocol from honest parties into the session key. This was also suggested for CPace by Björn Tackmann [Tac19]. This way any modification of messages by the adversary will result in different keys for the peers. Correspondingly some RAM memory is required for storing the transcripts. In the case of CPace we would like to avoid this and prove security also for a protocol not hashing the full transcript.

For this purpose, we need to prove the following lemma. Given two honest parties calculating a Diffie-Hellman shared secret point  $K = (B^{y_a})^{y_b} = Y_a^{y_b} = Y_b^{y_a}$  the only functions  $T_1$  and  $T_2$  mapping nontrivial generators  $Y_a$  and  $Y_b$  of  $\mathcal{J}$  to points on the curve such that  $(T_1(Y_a))^{y_b} = (T_2(Y_a, Y_b))^{y_a}$  is  $(T_1(Y), T_2(Y, Z) = (Y^f, Z^f)$  .

The proof is carried out as follows. The public points of honest parties  $Y_a$  and  $Y_b$  are generators of  $\mathcal{J}$  since an arbitrary power of a generator that is not the neutral element is again a generator. So there is a factor  $f$  such that  $Y_a^f = T_1(Y_a)$ . Consequently  $(Y_a^f)^{y_b} = (T_1(Y_a))^{y_b} = (T_2(Y_a, Y_b))^{y_a} = Y_b^{y_a}$ . When re-ordering the exponentiations we obtain  $T_2(Y_a, Y_b) = Y_b^f$ .

So if the adversary aims at "non-destructively" deliver tampered points  $T_1(Y_a)$  and  $T_2(Y_a, Y_b)$  (that are on the same curve as the honest points) instead of the honest parties' points, the only way is to apply a common exponent  $f$ .

It remains to be shown that the adversary may not provide points  $Z_a$  and  $Z_b$  on *another* curve (e.g. the twist) such that  $Z_b^{y_a} = Z_a^{y_b}$  iff  $Y_b^{y_a} = Y_a^{y_b}$ . Here again, we have information-theoretic security, similar to the reasoning for the `GuessK` . Note that in order to retrieve any information on the Diffie-Hellman point, the adversary needs some knowledge on the base point used by both parties. In both games, however, the environment has no information on the base points used. Note that the simulator could choose to use a new arbitrary base point  $B_{sid}$  for each new session for both, the simulation of the hash oracle  $H_1$  and the simulated honest party points in  $G_3$  .

As a result, we conclude that the protocol is secure even for the variant that does not hash the protocol transcript into the session keys *sk1*.

## 4.4 Proof that probability `GuessK` in $G_3$ is negligible

### 4.4.1 Passive adversary

In  $G_2$  an eavesdropping adversary has access to the published points  $Y_a$  and  $Y_b$  generated by honest parties. In  $G_2$  he could make guesses for the password and derive the corresponding generator  $G$  from the known *sid*. If the adversary was able to calculate  $K$  in  $G_2$  from known  $G$ ,  $Y_a$  and  $Y_b$  without knowing  $y_a$  and  $y_b$  she would be able to solve the computational Diffie-Hellman problem, which is excluded by assumption A0.

The functionality  $\mathcal{F}_{\text{RO}}$  proceeds as follows, running on security parameter  $k$  with parties  $P_1, \dots, P_n$  and an adversary  $\mathcal{S}$ :

$\mathcal{F}_{\text{RO}}$  keeps a list  $L$  (which is initially empty) of pairs of bit strings.

Upon receiving a value  $(sid, m)$  with  $(m \in \{0, 1\}^*)$  from some party  $P_i$  or from  $\mathcal{S}$ , do:

- If there is a pair  $(m, (\tilde{h}))$  for some  $\tilde{h} \in \{0, 1\}^k$  in the list  $L$ , set  $h := \tilde{h}$ .
- If there is no such pair, choose uniformly  $h \in \{0, 1\}^k$  and store the pair  $(m, h) \in L$ .

Once  $h$  is set, reply to the activating machine (i.e., either  $P_i$  or  $\mathcal{S}$ ) with  $(sid, h)$ .

**Figure 7:** Ideal functionality  $\mathcal{F}_{\text{RO}}$  .

#### 4.4.2 Active adversary

Now let us consider the case that the adversary did provide one of the points  $Y_a$  and  $Y_b$  while the other point was calculated from  $G$  by a honest party. The case is symmetric for client and server, so let us assume here that the adversary provided a chosen  $Y_a$ . We will base the analysis here on an information-theoretic reasoning.

$G_3$  can be distinguished by  $\mathcal{Z}$  from  $G_2$  iff the impersonating adversary succeeds in obtaining the real-world protocol's honest party point  $K = Y_a^{y_b}$  in  $G_2$  . This implies that the adversary is able to calculate  $K$  from the following set of information:  $Y_b$ , a chosen  $Y_a$  and a guessed  $G$ . Note that upon the unknown choice of the honest party  $y_b$  any point in the prime-order subgroup of  $\mathcal{J}$  could result for  $K$  with uniform probability. The amount of required information for calculating  $K$  is thus governed by the order of the prime-order subgroup  $q$  and corresponds, e.g. to  $\log_2(q)$  bits.

The only information that the adversary disposes of is the fact that for the correct password-derived generator  $G$  the equation  $Y_b = G^{y_b}$  holds. Let us consider an adversary  $\mathcal{A}'$  which knows the discrete logarithm of  $Y_b$  with respect to a base point  $B$ . I.e. she knows  $z$  such that  $Y_b = B^z$ .  $\mathcal{A}'$  is clearly more powerful than  $\mathcal{A}$  because she could calculate  $Y_b = B^z$ .  $\mathcal{A}'$  has now access to the equation  $g \times y_b = z$  which contains more information than  $Y_b = G^{y_b}$ . Still since  $g$  is unknown to the adversary by assumption A2,  $G$  could be any point in the image of  $\text{Map2Point}$  with close to uniform probability by assumption A2. This means that there is no significant amount of information that  $\mathcal{A}'$  could learn (at most  $\log_2(q/r)$  bits, where  $r$  is the number of points in the image of  $\text{Map2Point}$ ). This amount of information is not sufficient for calculating  $K$ .

Under the given assumption set, the probability  $\text{GuessK}$  is, thus, negligible. <sup>1</sup> This makes the real-world execution of the CPace protocol indistinguishable from the ideal world.  $\square$

### 4.5 Remarks regarding the ordering of the messages and efficiency

Note that in this proof we have assumed that the server party starts with the communication round. In fact, since the services provided to the two parties by the ideal functionality are identical and since the protocol is perfectly symmetric, we could interchange the server and client roles for the balanced PAKE sub-protocol CPace. This might be useful, specifically in case that the scalar multiplication takes comparable time as message delivery.

<sup>1</sup>Note that if  $\mathcal{A}$  knows  $g$  with  $G = B^g$  there is an obvious attack. When providing a chosen point  $Y_a = B$ ,  $K$  could be calculated as  $K = Y_b^{1/g}$ .

The functionality  $\mathcal{F}_{\text{apwKE}}$  is parametrized by a security parameter  $k$ . It interacts with an adversary  $\mathcal{S}$  and a set of parties via the following queries:

### Password storage and authentication sessions

**Upon receiving a query** (`StorePWfile`,  $sid, P_i, P_j, pw$ ) **from party**  $P_i$  or  $P_j$  :

If this is the first `StorePWfile` query, store password data record (`file`,  $P_i, P_j, pw$ ) and mark it uncompromised.

**Upon receiving a query** (`ClSession`,  $sid, ssid, P_i, pw$ ) **from party**  $P_i$  :

Send (`ClSession`,  $sid, ssid, P_j, P_j$ ) to  $\mathcal{S}$ , and if this is the first `ClSession` query for  $ssid$ , store session record ( $ssid, P_i, P_j, pw$ ) and mark it fresh.

**Upon receiving a query** (`SvrSession`,  $sid, ssid$ ) **from party**  $P_j$  :

If there is a password data record (`file`,  $P_i, P_j, pw$ ) then send (`SvrSession`,  $sid, ssid, P_i, P_j$ ) to  $\mathcal{S}$ , and if this is the first `SvrSession` query for  $ssid$ , store session record ( $ssid, P_j, P_i, pw'$ ) and mark it fresh.

### Stealing password data

**Upon receiving a query** (`StealPWfile`,  $sid$ ) **from adversary**  $\mathcal{S}$ :

If there is no password data record, reply to  $\mathcal{S}$  with "no password file". Otherwise do the following. If the password data record (`file`,  $P_i, P_j, pw$ ) is marked uncompromised, mark it as compromised. **If there is a tuple (`offline`,  $pw'$ ) stored with  $pw = pw'$ , send  $pw$  to  $\mathcal{S}$ , otherwise** reply to  $\mathcal{S}$  with "password file stolen".

**Upon receiving a query** (`OfflineTestPwd`,  $sid, pw'$ ) **from adversary**  $\mathcal{S}$ :

If there is no password data record, or if there is a password record (`file`,  $P_i, P_j, pw$ ) that is marked uncompromised, then store (`offline`,  $pw'$ ). Otherwise, do: If  $pw = pw'$ , reply to  $\mathcal{S}$  with "correct guess". If  $pw \neq pw'$ , reply with "wrong guess".

### Active session attacks

**Upon receiving a query** (`TestPwd`,  $sid, ssid, P, pw'$ ) **from adversary**  $\mathcal{S}$ :

If there is a session record of the form ( $ssid, P, P', pw$ ) which is fresh, then do: If  $pw = pw'$ , mark the record compromised and reply to  $\mathcal{S}$  with "correct guess". Otherwise, mark the session record interrupted and reply with "wrong guess".

**Upon receiving a query** (`SvrImpersonate`,  $sid, ssid$ ) **from adversary**  $\mathcal{S}$ :

If there is a session record of the form ( $ssid, P_i, P_j, pw$ ) which is fresh, then do: If there is a password data record (`file`,  $P_i, P_j, pw$ ) that is marked compromised, mark the session record compromised and reply to  $\mathcal{S}$  with "correct guess", else mark the the session record interrupted and reply with "wrong guess".

### Key Generation and Authentication

**Upon receiving a query** (`NewKey`,  $sid, ssid, P, key$ ) **from adversary**  $\mathcal{S}$ , where  $|key| = k$ :  
If there is a record of the form ( $ssid, P, P', pw$ ) that is not marked completed, then:

- If this record is compromised, or either  $P$  or  $P'$  is corrupted, then output ( $sid, ssid, key$ ) to  $P$ .
- If this record is fresh, there is a session record ( $ssid, P', P, pw'$ ),  $pw' = pw$ , a key  $key'$  was sent to  $P'$ , and ( $ssid, P', P, pw$ ) was fresh at the time, then let  $key'' = key'$ , else pick a random key  $key''$  of length  $k$ . Output ( $sid, ssid, key''$ ) to  $P$ .
- In any other case, pick a random key  $key''$  of length  $k$  and output ( $sid, ssid, key''$ ) to  $P$ .

Finally, mark the record ( $ssid, P, P', pw$ ) as completed.

**Upon receiving a query** (`TestAbort`,  $sid, ssid, P$ ) **from adversary**  $\mathcal{S}$ :

If there is a session record of the form ( $ssid, P, P', pw$ ) that is not marked completed, then:

- If this record is fresh, there is a record ( $ssid, P', P, pw'$ ), and  $pw' = pw$ , let  $b' = \text{succ}$ .
- In any other case let  $b' = \text{fail}$

Send  $b'$  to  $\mathcal{S}$ . If  $b' = \text{fail}$ , send (`abort`,  $sid, ssid$ ) to  $P$ , and mark record ( $ssid, P, P', pw$ ) completed.

**Figure 8:** Ideal functionality  $\mathcal{F}_{\text{apwKE}}$  for verifier-based PAKE with explicit authentication from [GMR06]. Note that we applied a single wording change (underlined) by replacing `Impersonate` with `SvrImpersonate` for making it more explicit that this message models impersonation of the *server*. Also we allowed the password for the server be registered by both, the server *and* the client (gray shaded extension). When omitting the cyan section, we obtain the "strong", pre-computation-attack resistant version  $\mathcal{F}_{\text{SaPAKE}}$ .

## 5 Proof for the augmented protocol AuCPace

### 5.1 Technical details

#### 5.1.1 Mitigations regarding user enumeration attacks

For carrying out the proof, we aim at re-using functionalities from previous papers wherever possible, specifically the ideal functionality  $\mathcal{F}_{\text{apwKE}}$  from [GMR06].

However, for our real-world protocol we could not use it as-is because  $\mathcal{F}_{\text{apwKE}}$  aborts in case that the server does not find a password entry in its file. In our protocol, we aim at keeping the information which users have a database entry as confidential as possible in order to mitigate user enumeration attacks. Therefore we continue with a random  $PRS$  string instead of aborting, thus concealing the information at least from the low-motivation and low-skill attacker. Note that full confidentiality for database contents could be realized only when using a server-specific "pepper" which would allow for rainbow tables. With a transmitted "salt" we also couldn't hide the events of password changes. We first considered re-phrasing  $\mathcal{F}_{\text{apwKE}}$  but finally decided to stick with the established functionality which aborts if no password database entry is available (in line with [GMR06, JKX18]).

#### 5.1.2 Adaptions to $\mathcal{F}_{\text{apwKE}}$ , *relaxed* $\mathcal{F}_{\text{apwKE}}$

The second aspect to consider is the following property of  $\mathcal{F}_{\text{apwKE}}$  from [GMR06]. It allows for a `TestPwd` query once if a session is not yet completed. As we will point out later, in case of a AuCPace variant with implicit authentication where  $T_a$  and  $T_b$  messages are dropped, we will need an extension of the functionality. Specifically, we will need to allow for one single `TestPwd` query also after a session key has been delivered to the honest party just as in [JKX18] in section 5.4.

For AuCPace without explicit key confirmation, the functionality  $\mathcal{F}_{\text{apwKE}}$  needs the following extension.

- **Upon receiving a query** (`TestPwd, sid, ssid, Pi, pw'`) **from**  $\mathcal{S}$ , if there is a record (`ssid, Pi, Pj, pw`) marked `completed` with session key  $sk$  and there is no record (`ssid, Pj, Pi, pw*`) that is also marked `completed` and this is the first `TestPwd` message for `ssid` and  $\mathcal{S}$ , do: if  $pw' = pw$ , return  $sk$  to  $\mathcal{S}$ ; else return "wrong guess".

We refer to  $\mathcal{F}_{\text{apwKE}}$  with this extension as *relaxed*  $\mathcal{F}_{\text{apwKE}}$ .

#### 5.1.3 Password hash

The second technical aspect to consider is the handling of the  $\text{PBKDF}_\sigma(pw, \text{username}, \text{salt})$  function. In the proof, we treat  $\text{PBKDF}$  as a separate hash function  $H_6$  and model it as a random oracle  $\text{PBKDF}_\sigma(pw, \text{username}, \text{salt}) = H_6(pw || \sigma || \text{username} || \text{salt})$ .

The third technical aspect stems from the fact, that the UC simulation model based on Turing machines does not naturally allow for the concept of human users with "user names" and authorizations. Instead we assume that the client's identifier  $P_i$  takes over the role of the user name and ignore the concept of authorization here. The full protocol as used for the proof is shown in figure 10.

#### 5.1.4 Keeping the clear-text password secret for the server

The fourth aspect is related to password registration. In the original formulation of  $\mathcal{F}_{\text{apwKE}}$  in [GMR06] the password is registered on a server by giving it the clear-text password  $pw$ . We have provided the extension that we also allow the password to be provided only to the client in clear-text form. Our version of  $\mathcal{F}_{\text{apwKE}}$  thus provides also for the

The functionality  $\mathcal{F}_{\text{tunnel}}$  provides the service of confidential and authentic message delivery from party  $P_i$  to  $P_j$ :

**Upon receiving a query**  $(\text{TunnelMsg}, \text{sid}, P_i, P_j, \text{data})$  **from party**  $P_i$ :  
 Send  $(\text{TunnelMsg}, \text{sid}, P_i, P_j, \text{data})$  to  $P_j$ .

**Figure 9:** Ideal functionality  $\mathcal{F}_{\text{tunnel}}$  as used for password registering in the real-world protocol.

option of registering the password on the server by a query that is issued by the client (gray shaded extension in figure 8).

In order to formalize the process of password registration in a setting where only the client has access to the clear-text password, we introduced the ideal functionality  $\mathcal{F}_{\text{tunnel}}$  in figure 9.

### 5.1.5 Difference between server corruption and server compromise

We adhere to the convention from [GMR06] and use "server compromise" for the event of stealing the server's persisted state. We use the terminology denote "corruption" for events where the adversary gains control over a party during session establishment.

## 5.2 Proof strategy for the augmented protocol

With respect to simulation, we need to distinguish password storage and session establishment. During password storage we proceed as in [GMR06] and do not actually give the adversary  $\mathcal{A}$  any power. In order to model this we have to add a message tunnel between the client and server parties in the real-world setting, as formally defined by the functionality  $\mathcal{F}_{\text{tunnel}}$ . In the ideal world, we use the gray-shaded extension to  $\mathcal{F}_{\text{apwKE}}$  from figure 8 by registering the clear-text client password directly from the client side.

With respect to the session establishment, we again consider fully-adaptive adversaries which are able to read and modify the messages. Just as for the proof of the balanced sub-protocol CPace, we proceed by using a sequence of games where  $G_0$  corresponds to the real world and  $G_4$  corresponds to the ideal world. In each of these games, we consider simulators  $\mathcal{S}_0$  to  $\mathcal{S}_4$  which implement part of their functionality in a subroutine library  $\mathcal{F}_0$  to  $\mathcal{F}_4$ , where  $\mathcal{F}_4$  exactly implements the ideal functionality  $\mathcal{F}_{\text{apwKE}}$ . Throughout this proof, we show that all of the individual games are indistinguishable for  $\mathcal{Z}$ .

## 5.3 Game-based proof

**Theorem 2.** *The protocol from figure 10 securely realizes  $\mathcal{F}_{\text{apwKE}}$  in the  $(\mathcal{F}_{\text{RO}}, \text{relaxed } \mathcal{F}_{\text{pwKE}}, \mathcal{F}_{\text{tunnel}})$  or  $(\mathcal{F}_{\text{RO}}, \mathcal{F}_{\text{pwKE}}, \mathcal{F}_{\text{tunnel}})$  hybrid models in the presence of adaptive adversaries under the assumption of the hardness of the computational Diffie-Hellman problem in  $\mathcal{J}$  (A3).*

As a result of the proof for the theorem above, we will also see that we could also prove security for an AuCPace protocol variant without explicit key confirmation. In this variant, no messages  $T_a$  and  $T_b$  are generated. Here  $P_i$  ( $P_j$ ) directly output the session key of steps 8 (9) in steps 6 (7) respectively. This variant securely realizes *relaxed*  $\mathcal{F}_{\text{apwKE}}$  in the  $(\mathcal{F}_{\text{RO}}, \text{relaxed } \mathcal{F}_{\text{pwKE}}, \mathcal{F}_{\text{tunnel}})$  hybrid model.

### 5.3.1 Game $G_0$ : Real Game

$G_0$  is the real game in the random-oracle model using the functionality  $\mathcal{F}_{\text{RO}}$  from figure 7 for calculating the password hash PBKDF. Honest parties  $P_i$  execute the actions of

### The Asymmetric AuCPace protocol

**Setup:** This protocol uses a random oracle functionality  $\mathcal{F}_{\text{RO}}$  for all of the hash functions  $\text{H}_3, \text{H}_4, \text{H}_5$  and the password hash  $\text{PBKDF}_\sigma(\text{H}_6)$  with a parametrization  $\sigma$  and salt size of  $m_s$  bits. The protocol also uses a balanced PAKE functionality  $\mathcal{F}_{\text{pwKE}}$  as well as a Diffie-Hellman key exchange protocol (written in exponentiation notation  $X^y$ ) operating on base point  $B$  and group order  $m_{\mathcal{J}}$  working on a cryptographic sub-group  $\mathcal{J}$  of an elliptic curve and its quadratic twist  $\mathcal{J}'$ . For password registration it uses the ideal functionality  $\mathcal{F}_{\text{tunnel}}$ .

#### Password storage protocol:

When  $P_i$  (who is a client) is activated using  $\text{StorePWfile}(sid, P_j, pw)$  for the first time, he does the following. He samples a fresh value  $\text{salt} \leftarrow_{\$} \{0, 1\}^{m_s}$ , calculates the password hash  $w = \text{H}_6(\text{salt} \parallel \sigma \parallel pw \parallel P_i)$  by using  $\mathcal{F}_{\text{RO}}$ . He then calculates a Diffie-Hellman point  $W = B^w$ .  $P_i$  then sends a query  $(\text{TunnelMsg}, sid, P_i, P_j, (\text{StorePWfileSvr}, \text{salt}, \sigma, W))$  to  $\mathcal{F}_{\text{tunnel}}$ .

When  $P_j$  which is a server receives  $(\text{TunnelMsg}, sid, P_i, P_j, (\text{StorePWfileSvr}, \text{salt}, \sigma, W))$  from  $\mathcal{F}_{\text{tunnel}}$  for the first time, he sets  $\text{file}[sid] = (sid, \text{salt}, \sigma, W, P_i)$ .

#### Protocol steps for session establishment:

1. When  $P_j$  receives input  $(\text{SvrSession}, sid, ssid, P_i)$ , he sets up a session record  $(sid, ssid, P_i)$  and marks it as fresh. He then waits for a  $(\text{username}, sid, ssid, P_i)$  message.
2. When  $P_i$  receives input  $(\text{CltSession}, sid, ssid, P_j, pw)$  he sets up a session record  $(sid, ssid, P_j)$  and marks it as fresh. He then sends message  $(\text{username}, sid, ssid, P_i)$  to  $P_j$  and awaits a response.
3. When  $P_j$  receives input  $(\text{username}, sid, ssid, P_i)$ , he obtains the tuple stored in  $\text{file}[sid]$  (aborting and marking the session record as **completed** if this value is not properly defined). He then samples a fresh nonzero exponent  $x$  with  $0 < x < m_{\mathcal{J}}$  and calculates  $X = B^x$  and  $W^x$ . He then continues with sending  $(\text{hashingParams}, sid, ssid, \sigma, \text{salt}, X)$  to  $P_i$  and waits for **continueAfterHashingParams**.
4. When  $P_i$  receives input  $(\text{hashingParams}, sid, ssid, \sigma, \text{salt}, X)$  he verifies  $X$  and calculates  $w = \text{H}_6(\text{salt} \parallel \sigma \parallel pw \parallel P_i)$ . He then calculates  $X^w$ . He then sends  $(\text{NewSession}, (sid, ssid), P_i, P_j, (sid, ssid, X^w))$  to  $\mathcal{F}_{\text{pwKE}}$  and awaits a response.
5. When  $P_j$  receives input  $(\text{continueAfterHashingParams}, sid, ssid, P_i)$  he sends  $(\text{NewSession}, (sid, ssid), P_j, P_i, (sid, ssid, W^x))$  to  $\mathcal{F}_{\text{pwKE}}$ .
6. When  $P_j$  receives input  $((sid, ssid), sk1)$  he calculates  $T_a = \text{H}_3(sk1), T'_b = \text{H}_4(sk1)$  and  $sk = \text{H}_5(sk1)$  and adds  $T_a, T'_b, sk$  to the session record. He then sends  $(\text{Authenticator}, sid, ssid, T_a)$  to  $P_i$  and awaits a response.
7. When  $P_i$  receives input  $((sid, ssid), sk1)$  he calculates  $T'_a = \text{H}_3(sk1), T_b = \text{H}_4(sk1)$  and  $sk = \text{H}_5(sk1)$  and adds  $T'_a, T_b, sk$  to the session record. Then he sends  $(\text{Authenticator}, sid, ssid, T_b)$  to  $P_j$  and outputs  $(sid, ssid, sk)$ . He then waits for a response.
8. When  $P_i$  receives a message  $(\text{Authenticator}, sid, ssid, T_a)$  he compares  $T'_a$  with  $T_a$  and aborts in case of differences. Else  $P_i$  outputs  $(sid, ssid, sk)$ .
9. When  $P_j$  receives a message  $(\text{Authenticator}, sid, ssid, T_b)$  he compares  $T'_b$  with  $T_b$  and aborts in case of differences. Else  $P_j$  outputs  $(sid, ssid, sk)$ .

**Stealing the password file:** When  $P_j$  (who is a server) receives a message  $(\text{StealPWfile}, sid, P_j, P_i)$  from the adversary  $\mathcal{A}$ , if  $\text{file}[sid]$  is defined,  $P_j$  sends it to  $\mathcal{A}$ .

**Figure 10:** AuCPace Protocol definition for the proof of indistinguishability.



the real-world protocol until eventually getting corrupted. Note that we need to let the adversary re-activate the server party with a `continueAfterHashingParams` message after receiving the `NewSession` response from  $\mathcal{F}_{\text{pwKE}}$  for the client. Specifically client entities  $P_i$  receive `StorePWfile` and `CltSession` queries from the environment  $\mathcal{Z}$  and return session keys upon success. Server entities  $P_j$  receive `SvrSession` queries. On the event of corruptions, all the internal state of the parties is passed to the real-world adversary algorithm  $\mathcal{A}$ , specifically for server corruptions, the password verifier  $W$  is returned. Honest parties use the API of  $\mathcal{F}_{\text{pwKE}}$  for generating their session keys and also the adversary is able to interface to  $\mathcal{F}_{\text{pwKE}}$  when impersonating peers. The subroutine library  $\mathcal{F}_0$  is empty.

### 5.3.2 Game $G_1$ : Modeling the random oracle for the hash

In  $G_1$  we replace calls to  $\mathcal{F}_{\text{RO}}$  by a separate implementation of the random oracle for PBKDF and the hash functions in a straight-forward way. Again we maintain an initially empty list  $\Lambda$  of value pairs  $(n, q, r)$ . For any hash query  $H_n(q)$  such that  $(n, q, r)$  appears in  $\Lambda$  from any of the ITI, the returned answer is  $r$ . In case that no query  $q$  has yet occurred we implement the conventional random-oracle model by choosing a new random  $r$  of length  $k$ , by storing  $(n, q, 0, r)$  in  $\Lambda$  and by returning  $r$  to the calling ITI.

This game is indistinguishable from game  $G_0$  since the hash results both come from the same uniform distribution.

### 5.3.3 Game $G_2$ : Getting rid of the case where the adversary $\mathcal{A}$ wins by chance.

This game is almost as the previous, only we allow the simulator to abort in case that the adversary manages to guess one of the authenticator messages  $T_a$  or  $T_b$  without querying the random oracles for  $sk1$ . This happens with negligible probability, so Game  $G_1$  and  $G_2$  are indistinguishable for  $\mathcal{Z}$ .

### 5.3.4 Game $G_3$ : Handle mutual authentication.

In this game we deal with mutual authentication. We will allow the simulator access to the clear-text password  $pw$  upon server compromise events only. I.e. we let  $\mathcal{F}_3$  implement  $\mathcal{F}_{\text{apwKE}}$  from figure 8 and additionally allow access to the clear-text password upon server compromise events.

This means that we have to simulate all of the protocol messages without knowledge of  $pw$  unless under corresponding conditions where the adversary has learned it when compromising honest parties in  $G_2$ .

First we consider how honest parties and the simulator behave for session key establishment and password registration.

Regarding password registration, honest clients will pass the password  $pw$  in the `StorePWfile` to  $\mathcal{F}_3$ . Note that our extended version of  $\mathcal{F}_{\text{apwKE}}$  also allows the *client*  $P_i$  to register passwords for a server  $P_j$ . This way, the simulator will not learn  $pw$ .

When a session is established, when a honest client initiates a session establishment in  $G_2$  for password  $pw$ , we let it issue a `CltSession` query to  $\mathcal{F}_3$ . This party will wait until it receives a session key. The simulator will have to arrange for simulating all subsequent messages that would have been issued by this party.

The simulator first transmits the `username` query to the server party. On the server side, a `SvrSession` query will be issued to  $\mathcal{F}_3$ . The party will wait subsequently for a session key to be delivered. The simulator has to provide all corresponding messages from  $G_2$  in an indistinguishable way.

After the `username` query, the first message to simulate is the `hashingParams` message. Here for each server *sid* we sample a new random value for the salt. We let the simulator also calculate a new DH key pair  $(x, X)$  and provide  $X$  in the `hashingParams` message.

Point verification for  $X$  on the client side and possible abortion by the client is implemented just as in  $G_2$ .

In the real-world, now  $\mathcal{F}_{\text{pwKE}}$  would be invoked by the client, which is not present for honest parties in  $G_3$ .

Subsequently, the simulator makes a calls to the **TestAbort** queries in order to simulate the explicit mutual authentication of both parties and issues **NewKey** queries in case of success only. As authenticator messages the simulator provides  $T_a$  and  $T_b$  as random strings.

Since  $T_a$  and  $T_b$  come from the same distribution in both games and both honest parties abort under the same conditions, this part of the simulation is indistinguishable from the real-world.

This way, honest parties did no longer use the API of the balanced PAKE  $\mathcal{F}_{\text{pwKE}}$  as in the previous game but use the functionality  $\mathcal{F}_3$ .

For an adversary that did interact with  $\mathcal{F}_{\text{pwKE}}$  in the previous game, the simulator receives all of the corresponding queries. Specifically, we will learn the input  $V$  to the **NewSession** queries which contain the values  $X^w$  and  $W^x$  in case of honest parties and also the input  $V$  of a possible **TestPwd** query. Subsequently the simulator will have to provide session keys  $sk1$  indistinguishable from the ones that  $\mathcal{F}_{\text{pwKE}}$  has provided in  $G_2$ .

Let us first consider the case of  $\mathcal{Z}$  trying to impersonate a client. First the user-name message will be issued. The honest server will invoke  $\mathcal{F}_3$  and the simulator provides the salt and  $X$ . If  $\mathcal{Z}$  follows the real-world protocol honestly, it will query the password hash for a password  $pw$  and calculate  $X^w$ . The simulator will be able to observe the password hash query and maintain a password candidate list for the pair of  $(salt, pw, w)$ . The simulator will execute a corresponding **OfflineTestPwd** query to  $\mathcal{F}_3$ . When  $\mathcal{Z}$  invokes a **NewSession** query of  $\mathcal{F}_{\text{pwKE}}$  it provides a point  $V$ , that the simulator also receives. The simulator now iterates through the password candidate list for the known salt value and checks for an entry with  $V = X^w$ . In case of a match, the simulator has learned the password. In case that there is no match, there is no password hash query from  $\mathcal{Z}$  that did yield a secret exponent  $w$  being consistent with the point  $W^x$  that a honest party would have calculated. With overwhelming probability, in  $G_2$  the inputs for  $\mathcal{F}_{\text{pwKE}}$  would have differed, because even with knowledge of the public points  $W, X, B$  the adversary would have had to solve the CDH problem in order to calculate  $W^x = X^w$  that is used by the honest peer.

Since the simulator now has learned the password, it could issue the **TestPwd** query from  $\mathcal{F}_3$  and find out whether the guess of  $\mathcal{Z}$  was correct. The simulator will provide intermediate session keys for both sides as simulated output of  $\mathcal{F}_{\text{pwKE}}$  which match or differ depending on the correctness of the guess. From these values,  $T_a$  and  $T_b$  will be calculated from hash queries.

For the honest server party, we implement checking of  $T_b$  just as in the previous game (and let the party abort, in case of failures). Note that the only chance for  $\mathcal{Z}$  that allows for providing a correct  $T_b$  is the hash query to  $H_4$  (see  $G_1$ ), which requires  $\mathcal{Z}$  to obtain  $sk1$  either by waiting for the session key from  $\mathcal{F}_{\text{pwKE}}$  or by using  $\mathcal{F}_{\text{pwKE}}$ 's **TestPwd** query.

In any case the simulator lets session key establishment fail in case that **TestPwd** queries were not issued before a  $T_b$  is provided by  $\mathcal{Z}$ , or alternatively  $\mathcal{Z}$  has received a session key output that was simulated in place of  $\mathcal{F}_{\text{pwKE}}$ . Since in this case the honest peer aborts, the server session of  $\mathcal{F}_3$  is never completed and we could always issue the **TestPwd** query of  $\mathcal{F}_3$  as a response to the need of simulating a **TestPwd** query for  $\mathcal{F}_{\text{pwKE}}$  in this case without need for an extension in  $\mathcal{F}_3$ .

Note that in case that no explicit mutual authentication step is present, as in a protocol variant with implicit authentication, we would issue session keys without explicit key confirmation. In this case the records in  $\mathcal{F}_3$  will reach the **completed** state and we would not be allowed to use the **TestPwd** query of  $\mathcal{F}_3$  as needed for the case of late adversarial

TestPw queries to  $\mathcal{F}_{\text{pwKE}}$  in  $\mathsf{G}_2$ . However we could simulate this case, if we allow for the *relaxed* version of  $\mathcal{F}_{\text{apwKE}}$  as defined above.

$T_a$  and  $T_b$ ,  $X$  and the salt values come from the same distributions as in  $\mathsf{G}_2$ . Authentication succeeds or fails in both games under the same conditions, making the games indistinguishable for this case.

When  $\mathcal{Z}$  tries to impersonate a *server* in  $\mathsf{G}_2$ , we proceed as follows. When we simulate a *username* message for the honest client, the simulator receives a pair of values for  $(\text{salt}, X)$  from  $\mathcal{Z}$  as *hashingParams*. Subsequently, the simulator notices when  $\mathcal{Z}$  uses the API of  $\mathcal{F}_{\text{pwKE}}$  with some point  $V$ . Again, we re-visit the password candidate list containing triples of  $(\text{salt}, pw, w)$  and try to find a value  $X^w$  being consistent with  $V$ . In case that we find a match, we learned the password and issue TestPw and upon success SvrImpersonate queries to  $\mathcal{F}_{\text{apwKE}}$ . From these queries, we know whether a session is to be succeeding or not and could provide consistent session keys for  $sk1$ .

For the simulation of  $T_a$  and  $T_b$  we proceed as in the case where  $\mathcal{Z}$  did take over the role of the client. Similarly, if  $\mathcal{Z}$  provides a value  $T_a$  without using either NewSession or TestPw from  $\mathcal{F}_{\text{pwKE}}$ , we let the honest party abort. This way the client party's state within  $\mathcal{F}_{\text{apwKE}}$  never reaches *completed* and we could also arrange for later queries to TestPw by  $\mathcal{Z}$  that we handle just as mentioned above and forward to  $\mathcal{F}_3$ .

In case of compromising a honest server's password file, we have to return password verifiers  $W$  in order to maintain indistinguishability with game  $\mathsf{G}_2$ . In game  $\mathsf{G}_3$  we do so, by retrieving the clear-text password  $pw$  from  $\mathcal{F}_3$  and by calculating the password verifier as  $B^w$  just as in the original protocol.

Note also that the adversary that has compromised a honest server and learned  $W$  cannot calculate  $W^x$  or  $X^w$  from the information set  $X, W$  and  $B$  in  $\mathsf{G}_2$ . In  $\mathsf{G}_2$  it could only succeed in providing matching inputs to  $\mathcal{F}_{\text{pwKE}}$  in case that an offline search has succeeded. In this case the adversary knows  $w$  and no longer needs to solve the CDH problem. Unless the offline search has succeeded, in  $\mathsf{G}_2$  different session keys will be obtained by  $\mathcal{A}$  than the honest party. This is exactly what happens also in  $\mathsf{G}_3$ .

Games  $\mathsf{G}_2$  and  $\mathsf{G}_3$  are indistinguishable for the environment. In both games, the client aborts, if the group order of the point  $X$  is small. The Diffie-Hellman points  $W^x$  and  $X^w$  match, thus, iff  $W$  has been calculated from  $B^w$  and  $X$  has been calculated from  $B^x$ . Therefore any modification of  $X$  by  $\mathcal{A}$  leads to different *PRS* inputs to  $\mathcal{F}_{\text{pwKE}}$  with overwhelming probability. I.e., with overwhelming probability the input to  $\mathcal{F}_{\text{pwKE}}$  matches if the passwords used for the StorePWfile request for the client match the one from the CltSession request. As a consequence the session keys returned by  $\mathcal{F}_{\text{pwKE}}$  match only if the very same passwords match. Verification of the authenticators in  $\mathsf{G}_2$  succeeds iff  $\mathcal{F}_{\text{pwKE}}$  returned the same session key to both parties. Upon any modification of the authenticator messages by  $\mathcal{A}$  the parties abort in both games.

### 5.3.5 Game $\mathsf{G}_4$ : Keeping the password secret also after server compromise

In this game we disallow the simulator to access the clear-text password upon server compromise events.

In this game, we add a "re-program-offline-query" list  $\Lambda_1$  to the implementation of the random oracle in addition to its list  $\Lambda$ . We change the implementation as follows. Upon a query  $q$  to the hash oracle, if no corresponding entry is found in  $\Lambda$ , we first parse salt value  $P_i, pw$  and  $\sigma$  from the query  $q$ 's encoding. For all entries  $(sid, P_j, w)$  contained in  $\Lambda_1$  we already did execute an OfflineTestPw query on  $\mathcal{F}_4$  for all  $P_j$  in case of a "correct guess" result, we program the record in  $\Lambda$  for the query to the value  $w$  from  $\Lambda_1$ , remove the entry  $(sid, P_j, w)$  from  $\Lambda_1$  and return  $w$ . If after parsing the full list  $\Lambda_1$  no "correct guess" result is returned, we sample a fresh random value  $r'$ , program it to  $\Lambda$  and return  $r'$ .

Upon server compromise, we proceed as follows. We first make a StealPWfile query to  $\mathcal{F}_4$ . Subsequently, we iterate through the PBKDF's random oracle list entries in  $\Lambda$ , parse the

stored queries for the client id, salt,  $\sigma$  and the password and execute `OfflineTestPwd` queries to  $\mathcal{F}_4$ . In case of a "correct guess" reply, we learned the password  $pw$  and can, thus, calculate the password verifier  $W$  and send it to the adversary  $\mathcal{A}$ . Otherwise, the password hash oracle has not yet been queried for the specific password. In this case, we sample a new random hash result  $w$ , setup a new re-program-offline-query entry  $(sid, P_j, w)$  for the hash oracle  $\Lambda_1$ . In this case we calculate the password verifier as  $W = B^w$  and send it to the adversary.

This procedure allows us to later on arrange for matching password verifiers  $W$  and password hashes  $w$ .

Since we have now access to both,  $W$  and  $w$  we could simulate the case of server impersonation just as in the previous game.

The only difference to game  $G_3$  shows up with respect to the way that the password verifier  $W$  is calculated. Irrespectively, whether the adversary had queried the hash oracle before the server compromise operation or after, the simulator always returns  $(w, W)$  pairs matching to the respective passwords. Also in both games the distribution of password hashes and verifiers  $w$  and  $W$  is the same. Game  $G_3$  and  $G_4$  are, thus, indistinguishable for the environment  $\mathcal{Z}$ .

The real-world protocol AuCPace, thus emulates the ideal functionality  $\mathcal{F}_{\text{apwKE}}$  in the (*relaxed*  $\mathcal{F}_{\text{pwKE}}$ ,  $\mathcal{F}_{\text{RO}}$ ,  $\mathcal{F}_{\text{tunnel}}$ ) or  $(\mathcal{F}_{\text{pwKE}}$ ,  $\mathcal{F}_{\text{RO}}$ ,  $\mathcal{F}_{\text{tunnel}}$ ) hybrid models. (Irrespectively whether the  $\mathcal{F}_{\text{pwKE}}$  functionality used in the real-world setting allows (*relaxed*  $\mathcal{F}_{\text{pwKE}}$ ) or disallows ( $\mathcal{F}_{\text{pwKE}}$ ) for late `TestPwd` queries, the resulting protocol emulates  $\mathcal{F}_{\text{apwKE}}$ .)  $\square$

## 6 Partial augmentation

### 6.1 The ideal functionality $\mathcal{F}_{\text{papwKE}}$ for modeling *partial* augmentation.

For the proof we introduce a new concept for *partial* augmentation of a PAKE protocol. The corresponding functionality is depicted in figure 11. In comparison to  $\mathcal{F}_{\text{apwKE}}$  partial augmentation ( $\mathcal{F}_{\text{papwKE}}$ ) gives the attacker the possibility to also impersonate the *client* after having succeeded in compromising the server.

In the partially augmented variant of our protocol, we replace the server-chosen ephemeral key-pair  $(x, X)$  by a long-term key pair that is re-used over several login sessions (same *sid*, different *ssid*). We would have liked to choose the key-pair only once at the point, where the server's Turing machine is first instantiated and not upon each password configuration. Unfortunately this is technically not possible in the UC framework, since this would correspond to a shared state over several *sid*. For this reason, we need to let the server choose  $(x, X)$  upon password configuration and store  $W^x$  together with  $X$  in the password file, i.e., just as the password verifier, the public key  $X$  becomes part of the shared state for session *sid*.

After receiving the password verifier  $W$  through the tunnel, the server calculates the ephemeral pair  $(x, X)$  and the point  $W^x$  and records the pair of  $(X, W^x)$  what we call *personalization* of  $W$  for the specific server. The server then may discard  $x$ . Note that using a long-term public key essentially halves the computational complexity of AuCPace for the server for the case of the login sessions.

At a first glance, since after a server  $P_j$ 's compromise, none of the security guarantees with respect to the adversary are maintained, it might be argued that  $\mathcal{F}_{\text{papwKE}}$  does not actually provide any meaningful advantage in comparison to  $\mathcal{F}_{\text{pwKE}}$ .

The advantage, however, becomes obvious when considering the IIoT setting with more than two servers sharing the same user credentials. In fact after executing a `StealPWfile` query on server  $P_j$  the adversary has full control over  $P_j$ . Note, however that

The functionality  $\mathcal{F}_{\text{papwKE}}$  is an extension to the functionality  $\mathcal{F}_{\text{apwKE}}$  from figure 8. It implements all of the  $\mathcal{F}_{\text{apwKE}}$  queries and extends the capabilities of the adversaries by the following query:

**Upon receiving a query** ( $\text{CltImpersonate}, sid, ssid$ ) **from adversary**  $\mathcal{S}$ :

If there is a session record of the form  $(ssid, P_i, P_j, pw)$  which is *fresh*, then do: If there is a password data record  $(file, P_i, P_j, pw)$  that is marked *compromised*, mark the session record *compromised* and reply to  $\mathcal{S}$  with "correct guess", else mark the the session record *interrupted* and reply with "wrong guess".

**Figure 11:** Ideal functionality  $\mathcal{F}_{\text{papwKE}}$  for partial verifier-based PAKE with explicit authentication.

the adversary is *not* given the clear-text password  $pw$  from  $P_j$  upon server compromise. He is only granted the capability to execute `OfflineTestPwd` queries.

In settings where the adversary may expect other server entities  $P_k$  to operate with the same password  $pw$  as  $P_j$ , client impersonation for connections with  $P_k$  is still precluded.

Note that this is occurring exactly in the use-case of industrial control plants. There user credentials (password verifiers) may be shared by many small server entities, which may be comparably easily stolen/compromised. In this setting, server compromise might most likely be implemented by invasive attacks on the hardware, e.g., by stealing a first server, un-soldering microcontroller or memory chips and by side-channel attacks that re-open debug ports. In this setting  $\mathcal{F}_{\text{papwKE}}$  provides very meaningful protection to the honest subset of servers. It might be likely to detect theft of the device and the partial augmentation feature might provide a sufficiently large time-window allowing for changing user credentials on the plant.

Also undetected re-insertion of a compromised server in a plant may not be a relevant attack scenario, such that the additional capability of the adversary to impersonate the client on this specific server does not actually degrade the security in practice. Moreover, as we will show, the AuCPace scheme allows for a server-specific configuration for partial and full augmentation. A server entity where non-invasive attacks allowing for a re-insertion into an installation should be considered feasible might choose to implement  $\mathcal{F}_{\text{apwKE}}$  using AuCPace with ephemeral key pair  $(x, X)$  while a server where a more invasive attack is presumed necessary in order to compromise the database (leading to device destruction) might choose to use a long-term secret  $x$  and as a consequence  $\mathcal{F}_{\text{papwKE}}$ .

Similar security guarantees of  $\mathcal{F}_{\text{papwKE}}$  could also be realized if any server uses a different "salt" value for each client, e.g., by letting the server provide a random salt value upon password configuration. This, however precludes mechanisms offering an off-line user credential distribution because all server identities need to be known at the time when the user configures his password. It would not allow for the flexibility to asynchronously add further servers to a plant after password registration. Note also that this way upon password changes, the complex  $\text{PBKDF}_\sigma$  password hash would have to be calculated once for each server, significantly reducing the feasible strength of the workload parameters  $\sigma$ .

For the same reason "personalizing" a password hash for a server by hashing it together with the server ID provides weaker security guarantees than partially augmented AuCPace. Either the capability to add new servers to a plant after password registration is lost or a central password distribution server would be required to hold information allowing for impersonating any user. In the AuCPace context the distribution server would only hold information on  $W$  not allowing for impersonation attacks without offline dictionary attacks because entering the protocol in client role requires the password-derived scalar  $w$ .

## 6.2 Proof

**Theorem 3.** *The protocol from figure 10 with using a long-term key-pair  $(x, X)$  instead of the ephemeral key pair from step 3 in figure 10 securely realizes  $\mathcal{F}_{\text{papwKE}}$  in the  $(\mathcal{F}_{RO}, \mathcal{F}_{\text{pwKE}})$  hybrid model in the presence of adaptive adversaries under the assumption of the hardness of the computational Diffie-Hellman problem in  $\mathcal{J}$  and  $\mathcal{J}'$ .*

We implement the proof for the partially augmented protocol in the UC hybrid model, just as for the fully augmented variant.

Since we assume that the key pair  $(x, X)$  is used for several protocol runs, we give the adversary access to the secret exponent  $x$  and the value  $W^x$  upon server compromise. For this reason, we also have to consider adversaries  $\mathcal{A}$  which base their attack strategy on this knowledge. We detect this case, that the adversary was giving a password related string  $PRS$  containing  $W^x$  as input to the balanced ideal functionality  $\mathcal{F}_{\text{pwKE}}$ . In case of such an attack strategy, we let the simulator use the `CtlImpersonate` query of  $\mathcal{F}_{\text{papwKE}}$  in order to make the ideal and real world indistinguishable for the environment  $\mathcal{Z}$ .  $\square$

## 7 Performance assessment of the AuCPace protocol

In this section, we will discuss the properties of AuCPace and other V-PAKE constructions from the perspective of extremely resource-constrained servers for industrial installations.

Fairly benchmarking algorithms and protocols often is a difficult task since many aspects need to be considered. Advantages regarding one side could often be obtained only at the cost of disadvantages elsewhere.

Since a large number of PAKE protocols have previously been suggested (see, e.g., [SOAA15, PW17] for a larger overview), we saw the need to focus here on a reduced subset of constructions considered suitable according to clear criteria. Our findings are summarized in table 1. For this discussion we concentrated on protocols fulfilling the following constraints.

- We restricted our analysis to constructions coming with explicit security proofs.
- We concentrated on constructions requiring "few" exponentiations because in our setting everything but the most efficient constructions were considered impractical. Essentially this requirement is equivalent to only considering constructions proven to be secure in the random oracle model. For the very same reason, we did not consider protocols that mandatorily require comparably costly large-characteristic field operations or pairings (or were only proven secure in this setting).
- We only considered V-PAKE schemes because we believe server compromise to be a highly relevant attack, specifically for plants with large outdoor installations. In this case in our opinion the additional mitigations provided by V-PAKE could prove beneficial, particularly when considered in conjunction with strong password hashes and password policies enforcing minimum entropy levels.
- We also considered proven forward secrecy to be mandatory for future-proof concepts since in important settings the secure channel established by the V-PAKE protocol will be used for securely changing passwords.

Specifically the requirement of proven forward secrecy turned out to be quite restrictive. Unfortunately many otherwise very interesting constructions, such as notably AugPake [SKI10] and SPAKE2+ [CKS08] do not offer this feature [PW17].



**Table 1:** Comparison of different V-PAKE constructions with explicit key confirmation. **Update July 2019:** Pre-computation attack resistance ("strong AuCPace", see appendix C) could be incorporated into both, partial and fully augmented AuCPace at the expense of additional  $+1v$  for the server and  $+2v+1i$  for the client and with  $\approx 16$  bytes larger password verifier sizes and communication bandwidth.

	AuCPace (part.)	AuCPace	VTBPEKE	OPAQUE
message count	4	4	3	3
message count pw-Registr.	1c	1c	1c	1s + 2c
precomp. res.	optional	optional	no	yes
proof	UC	UC	BPR(ROR)	UC
comp. complexity server	2v	3v+1f	3v+1f+1i	3v+1f
comp. complexity client	3v	3v	3v+1f	4v+1f
$x$ -coordinate only	possible	possible	-	-
simplified point ver.	possible	possible	-	-
pw-verifier size estimate	$\approx 96B$	$\approx 64B$	$\approx 64B$	$\approx 280B$
total message size estimate	$\approx 160B$	$\approx 160B$	$\approx 160B$	$\approx 280B$
Map2Point necessary	yes	yes	no	yes

## 7.1 Discussion of the comparison overview

In table 1 we summarize our results. For a fair comparison, the message count does not include the initial messages for session-id generation. Note that session id agreement is important for all of the UC-secure protocols. Moreover we only consider variants providing explicit mutual authentication.

Message counts for password registration consider the recommended procedure from the respective authors. The computational complexity comparison distinguishes fixed (f) and variable-point (v) scalar multiplications as well as group order field inversions (i).

The size estimate of password verifiers to store in the server's database assumes 32 bytes (B) for each of username, associated data for granted user authorizations and salt and compressed elliptic curve points for the 128 bit security level. We estimated 32 byte storage overhead for the overhead for symmetric authenticated encryption.

## 7.2 Security guarantees

While AuCPace and OPAQUE were both analyzed within the UC framework, the security proof for VTBPEKE relies on the BPR model and derives somewhat more quantitative upper bounds for the possible adversary advantage.

To some extent, the security guarantees depend on the framework used for the security analysis. For instance, unlike for the BPR-model, in the UC model used here and in [GMR06, JKX18] no assumptions regarding password distributions apply.

One minor difference might be that for AuCPace, we did consider fully adaptive adversaries during session establishment, while the other UC-based approaches [GMR06, JKX18] only considered static corruptions.

However, according to our assessment in the view of practical relevance for real-world applications, we believe that these differences among the filtered candidates within table 1 might be of somewhat minor importance. Among these four constructions, we only identified two significant security guarantee differences.

Firstly, one important feature of OPAQUE is that it allows for starting with the offline attack only *after* compromising the server (pre-computation attack resistance). With respect to this feature OPAQUE, provides stronger guarantees than the performance-optimized AuCPace versions analyzed in the previous sections.

**Update July 2019:** As an update of the TCHES paper version, this paper shows in appendix C how to add this security feature to both, partially and fully augmented AuCPace. We call the resulting protocol "strong AuCPace".

The unique feature of AuCPace is that it optionally allows for partial augmentation allowing for significantly less computational complexity than the other candidate protocols. This comes at the cost that in case of a corrupted server, the security guarantees correspond only to those of a balanced PAKE, while the V-PAKE guarantees are only maintained for other, yet uncompromised servers, sharing the same password database.

### 7.3 Computational efficiency for constrained servers

When considering the fully augmented setting all of the protocols in table 1 require four exponentiations per session for the constrained server. In case of OPAQUE and the fully augmented AuCPace one of these is a fixed-base point operation and could be pre-computed. The perceived delay on the HMI interface due to the complex scalar multiplications will correspond, thus, to only three scalar multiplications in the case of AuCPace and OPAQUE in contrast to four in the case of VTBPEKE.

It is worth noting, that an important part of the efficiency of OPAQUE could be attributed to the use of the highly efficient HMQV [Kra05] construction. Unfortunately HMQV is covered by patents in some important countries. When replacing HMQV with an alternative AKE not covered by patents, such as, e.g., NAXOS [LLM07], a corresponding OPAQUE-like construction would likely require five exponentiations. The problem of HMQV patents has been identified and possibly in the future HMQV might be freely usable [Kra18] for OPAQUE at least for some applications but, unfortunately, this is presently still uncertain.

We believe that many applications might be scared away from algorithms using patented components, not only because of the cost but also when considering complex licensing agreements and reporting duties.

For important applications in the IIoT setting, we conjecture that adequate security could be obtained also when implementing partial augmentation. Specifically here we consider settings where server compromise involves stealing of hardware and highly invasive physical attacks likely to destroy hardware.

In the partially augmented setting, AuCPace has a computational complexity of two exponentiations and provides the most efficient solution among all known verifier-based PAKE protocols. Attacks on yet un-compromised servers are prevented even if these are working with the same password verifier  $W$  as, e.g., distributed by a centralized database server.

### 7.4 Implementation effort

Both, OPAQUE and VTBPEKE require, in contrast to AuCPace, a full group structure for actual implementation, i.e., both,  $x$  and  $y$  coordinates of points are necessary and implementations have to deal with the point compression and point verification issues (and possibly patent trouble) if they aim at reducing the message and/or password verifier length.

AuCPace, in contrast could also be implemented by using  $x$ -coordinate-only Diffie-Hellman algorithms such as X25519. This could be used for saving both, code-ROM and RAM memories and could facilitate secure (e.g., efficient constant-time) implementation.

While beside the elliptic-curve operations AuCPace only requires a cryptographic hash, the other protocols also all require a symmetric encryption primitive. AuCPace is, thus, somewhat simpler to implement. This, however, might be only a rather minor advantage in many settings, since most application based on PAKE protocols might anyway require symmetric authenticated encryption after successful generation of the session key.

VTBPEKE and constructions based on [GMR06] have the advantage that they could be implemented without requiring hashing to elliptic curve points. Note that in the light of intellectual property rights, this could be an advantage, specifically regarding standardized curves in short-Weierstrass form. For instance for the application in travel documents the PACE protocol [BFK09] often is implemented in it's computationally complex "generic mapping" variant. The reason for this approach, in our opinion, is the fact that this avoids any need of hashing to curve groups and thus the mere risk of patent conflicts. Note that regarding patents it is, in our opinion, not the main question whether a method is actually rightfully covered by a patent or not. In fact already uncertainty and the merely potential risk of conflicts (and possibly large costs), might often motivate users to select circumvention techniques.

For VTBPEKE, unlike AuCPace, the server also needs to implement inversions with respect to the group order, i.e., in addition to the field arithmetic a second set of modulo reductions needs to be implemented.

**Update July 2019 :** In the case of strong AuCPace with pre-computation attack resistance, the field inversion operation is only required for the client side, which often could be assumed to have more resources.

AuCPace, thus, allows in comparison to the other candidates for significantly improved ease-of-implementation. This possibly might reduce the risk of implementation errors.

## 7.5 Bandwidth and latency aspects

At a first glance, AuCPace requires a comparably large number of communication rounds. When considering also the initial establishment of the (sub-) session id (*ssid*) by exchange of the messages  $t$  and  $s$ , we come up with a total message count of 5. Without counting the *ssid* establishment, the message count amounts to 4 messages only.

Both OPAQUE and VTBPEKE require 3 messages. However in the case of OPAQUE this does not account for messages required for session id establishment. It is worth noting that according to our best knowledge and in line with the conclusions from the analysis of Fischlin, Bender and Kügler [BFK09], the security model of the UC framework assumes mandatorily that the *ssid* is fixed prior to initiating the protocol. Specifically, this was identified as one obstacle preventing security analysis of PACE in an UC context. The requirement of session id establishment should in fact also apply for other UC-secure constructions such as OPAQUE and protocols based on [GMR06]. However unlike here the corresponding message count is not considered in the respective papers. Note that for AuCPace, actually the first usage of the *ssid* occurs only when entering the CPace protocol steps.

We believe that the typical use-case of a PAKE protocol is establishment of a secure (encrypted and authenticated) channel using the session key. In this case, the final two authenticator messages are optional, just as for OPAQUE, reducing the message count to 3 and 2 respectively for AuCPace and OPAQUE when disregarding session id establishment. Note that for both, OPAQUE and AuCPace the two explicit authentication messages are not mandatory for UC-securely implementing the  $\mathcal{F}_{\text{apwKE}}$  functionality.

In comparison to AuCPace, OPAQUE and VTBPEKE require one and two messages less, respectively, when considering explicit session id generation rounds. As a result, message latency will add up a bit more for AuCPace. However, in comparison to OPAQUE and [GMR06], messages used for AuCPace and VTBPEKE are significantly shorter. Specifically, it is not necessary to transfer encrypted (and authenticated) versions of public-private

key pairs. Note that this provides an advantage when using the PAKE protocol over a low-bandwidth wireless link, specifically if very small packets are used on the physical layer, such as in case, e.g., for the bluetooth-low-energy standard. Note also, that the shorter messages allow for reduced buffer sizes and all-over reduced RAM memory requirements.

AuCPace allows for pipelining message transfer and cryptographic calculation, improving upon user-experienced latency. For instance, the server may interleave transmission of the last message from the augmentation layer ( $X, \sigma, \text{salt}$ ) and calculation of the public point  $Y_a$ , such that  $Y_a$  is transmitted later in a separate message. In settings where message delivery latency is significant and computation is fast, however, the server may choose to include  $Y_a$  in the earlier message.

## 7.6 Intellectual property rights

We believe that pending patents on algorithms and algorithmic substeps might seriously hamper actual use of a cryptographic protocol. AuCPace was specifically designed for avoiding all patents known to the authors. We believe that this also applies to VTBPEKE. The only aspect where we are aware of the potential of conflicts for AuCPace is the Map2Point substep where efficient algorithms might possibly have to be analyzed in detail, specifically for curves in short Weierstrass form. For this reason, we will sketch a circumvention approach in the appendix A.

## 7.7 Registering verifiers

Regarding password verifier registration on the server we identify two main aspects: Size of the password verifiers on the server and flexibility regarding password-verifier registration protocols.

AuCPace is characterized by requiring only very little persistent storage for the password verifiers, i.e., in addition to the user identifiers, the encoding of the user's authorization and the salt value (which might require roughly 32 bytes), only one (two) group elements requiring typically 32 bytes need to be stored for full (partial) augmentation. In total an amount of 64 (96) bytes suffices. (Possibly further future analysis, e.g. in the BPR model could reduce the verifier size for partially augmented variants also to 64 bytes.)

Other protocols, specifically [GMR06, JKX18] require significantly longer verifiers. For instance, OPAQUE requires seven group elements or secret scalars, three of which are included in an encrypted authenticated structure (typically requiring additional nonce and MAC fields with, e.g., 2x16 bytes). Even when considering point compression, this could easily add up to a total verifier size of  $> 280$  bytes.

Note that the size of password verifiers should not be disregarded. Some microcontrollers include small amounts (e.g., 1 kByte) of somewhat protected memory (e.g., tamper-protected RAM) meant to be used for storing sensitive information such as cryptographic keys or password-related information. Excessive size of password verifiers might require additional complexity in the application code or make implementers be tempted to use unprotected memory also on devices that might be exposed to physical attacks. Note also that write operations on persistent memory could generate large power consumption transients.

In our opinion, the fact that AuCPace only needs fairly short password verifiers is strongly linked to the fact that it is a sequential scheme executing Diffie-Hellman before the SPEKE-like substep and not in parallel. In schemes using a large amount of parallel operations, such as for instance SPAKE2+ [CKS08] or OPAQUE [JKX18] much more complex password verifiers have to be used.

Regarding OPAQUE, it is worth to draw attention to a side-effect of the suggested process for password registration in [JKX18], because it is suggested to establish a bi-directional online connection to the client upon password changes. Note that this would

make user credential distribution protocols based on offline tickets (allowing for only one unidirectional message) impossible.

## 8 Implementation on ARM Cortex M0 and M4 microcontrollers

One important target platform for resource-constrained (I)IoT devices are 32 bit microcontrollers, such as the ARM Cortex M0 and Cortex M4 series. We have implemented AuCPace25519 in its partially augmented variant on nRF51 and nRF52 microcontrollers from the company Nordic Semiconductors and three different microcontrollers from the manufacturer ST Microelectronics.

In this section we first describe the high-level strategy for implementing the X25519 Diffie-Hellman protocol and Elligator2. The method that we employ for Elligator2 has already been presented in [BDL<sup>+</sup>11, BHKL13], but we considered it helpful to write it down explicitly here, because we were aware of it only after having received a corresponding hint [Ham17]. Subsequently, we will elaborate on our strategy for implementing the field arithmetic.

### 8.1 Implementation of X25519

AuCPace25519 uses X25519 for generating the password verifier and the session key  $sk_1$ . We make use of the constant-time Montgomery ladder algorithm from [DHH<sup>+</sup>15]. Both fixed-point and variable-point scalar multiplication require 1287 field multiplications and 1274 field squarings. We did implement two variants. Firstly for the sake of comparison with related work, we implemented a synchronous version of the X25519 function. Secondly, we implemented a second, asynchronous version of X25519. For the latter we implemented an asynchronous cryptographic engine (ACE) object that stores the intermediate state of the scalar multiplication. This allows our implementation to suspend and resume calculations after each ladder step in case that the power budget requires the microcontroller to enter a sleep mode.

We came to the conclusion, in line with findings from [Ham12], that the constant-time Montgomery ladder is, most probably the most efficient known algorithmic choice available for Diffie-Hellman on Curve25519, if we aim at avoiding memory consuming pre-computed tables.

### 8.2 Implementation of Elligator2

In order to remain consistent with the notation used in [BHKL13] we denote the Legendre symbol that records quadratic residuosity of  $a \bmod q$  (with  $q$  being an odd prime number) with  $\chi$ :

$$\chi(a) \triangleq \left(\frac{a}{q}\right) \equiv a^{\frac{q-1}{2}} \quad (1)$$

Remember that the Elligator2's decoding function for a Weierstrass curve  $E : y^2 = x^3 + Ax^2 + Bx$  is the function  $\psi : R \rightarrow E(\mathbb{F}_q)$  defined as follows:  $\psi(0) = (0, 0)$ ; if  $r \neq 0$  then  $\psi(r) = (x, y)$  (see [BHKL13]). For a set  $R$  defined as

$$R \triangleq \{r \in \mathbb{F}_q : 1 + ur^2 \neq 0, A^2ur^2 \neq B(1 + ur^2)^2\} \quad (2)$$

the following elements of  $\mathbb{F}_q$  are defined (we use only  $x$  coordinates):

$$v = \frac{-A}{1 + ur^2} \quad (3)$$

$$\varepsilon = \chi(v^3 + Av^2 + Bv) \quad (4)$$

$$x = \varepsilon v - (1 - \varepsilon) \frac{A}{2} \quad (5)$$

In case of Curve25519,  $q = 2^{255} - 19$ ,  $A = 486662$  and  $B = 1$ . Also  $u$  is chosen to be 2. If we would calculate  $x$  directly, we would need two exponentiations, one for the inversion (3) and one for the Legendre symbol  $\chi$  (4). In fact computing a single exponentiation is enough, when using the inverse square root algorithm from [BDL<sup>+</sup>11, BHKL13]. When substituting  $v$  in  $v^3 + Av^2 + Bv$  one obtains, in a projective representation, the fraction

$$\frac{a}{b} \triangleq \frac{A^3 ur^2 + AB(1 + ur^2)^2}{(1 + ur^2)^3} \quad (6)$$

As a property of the Legendre symbol we have:

$$\chi\left(\frac{a}{b}\right) = \chi(ab) \quad (7)$$

since  $\chi\left(\frac{a}{b}\right) \equiv a^{\frac{q-1}{2}} b^{-\frac{(q-1)}{2}} \equiv a^{\frac{q-1}{2}} b^{\frac{q-1}{2}} \equiv \chi(ab)$  with  $1 \equiv a^{q-1} \pmod{q}$  (Fermat's little theorem). Now let's define

$$c \triangleq ab \quad (8)$$

$$d \triangleq 1 + ur^2 \quad (9)$$

$$s \triangleq (cd^2)^{\frac{q-3}{2}} \quad (10)$$

We have  $s^2 cd^2 = (cd^2)^{q-2} = \frac{1}{cd^2}$  using Fermat's little theorem again. So the inverse is given by:

$$\frac{1}{d} = s^2 cd^2 cd \quad (11)$$

iff  $c \neq 0$  and  $d \neq 0$ . If  $c = 0$ , then the point is  $(0, 0)$  or  $\infty$ . If  $d = 0$  the point is  $\infty$  and we return 0 in either case. On the other hand it could be written  $s cd^2 = (cd^2)^{\frac{q-1}{2}} \equiv \chi(cd^2)$ . Furthermore it holds that by definition of the Legendre symbol:

$$\chi(ab) = \chi(a)\chi(b) \quad (12)$$

and

$$\chi(d^2) = 1 \quad (13)$$

unless  $d = 0$ . So

$$s cd^2 \equiv \chi(cd^2) = \chi(c) = \chi(ab) = \chi\left(\frac{a}{b}\right) \quad (14)$$

By placing (14) into (11) we get

$$\frac{1}{d} = \chi\left(\frac{a}{b}\right) s cd \quad (15)$$

This means that the Legendre symbol and the inverse (and finally Elligator2) was calculated by means of a single exponentiation (10). In case of Elligator2 for Curve25519 the algorithm requires a total of 254 field squarings and 23 field multiplications.

### 8.3 Implementation of the field arithmetic

The implementations for Cortex M0 and M4 share all of the group arithmetic and high-level algorithms but rely on separate field arithmetic for addition, subtraction, negation, multiplication and squaring.

Our implementation for the Cortex M0 uses the same highly optimized field arithmetic as in [DHH<sup>+</sup>15, HL17]. Here register pressure and the limited capability of the multiplier engine providing only 32 bits of a result make it beneficial to employ three cascaded Karatsuba stages. The implementation for the Cortex M4 makes use of a new, yet unpublished optimized implementation. Just as for the M0, we use a packed radix 32 representation. Throughout the implementation we reduce field elements modulo  $2^{256} - 38$ .

Our implementation on the M4 uses the much more powerful UMLAL and UMAAL instructions allowing for simultaneously multiplying two 32 bit words and accumulating up to two 32 bit words. Due to the significant advantage of using several stages of Karatsuba multiplication for the M0, we have implemented several variants of Karatsuba multiplication also for the Cortex M4. Experiments, however, have shown that here the reduced register pressure (the "upper" registers R8 ... R12 and R14 may be used without restrictions) in addition to the fact that accumulation comes essentially for free made schoolbook multiplication faster when register-allocation is carefully tuned. Based on our experiments, we presume that for carefully optimized code on the M4, Karatsuba techniques might become beneficial again for integer sizes above 512 bits.

In order to optimize the register allocation, we have generated the assembly sources by a code generator handling register allocation and spill register storage on the stack. For the accumulation of intermediate results during the multiplication we also make use of the UMLAL and UMAAL instructions. Note that when one register with the value 1 is available, UMAAL allows for implementing three 32 bit additions in one single cycle ( $r := a + b + 1 * c$ ) yielding a 64 bit result. For subtraction (and for squaring), we made use of a specific architectural property of the M4 architecture. There two distinct ways of handling addition carries are possible. In addition to the UMAAL -based method above, flag-based add and add-with-carry (ADDS, ADCS) and subtract-with-borrow (SUBS, SBCS) instructions are available. The multiplication instructions are specified not to modify the addition/subtraction carry flag. We did use this for merging reduction with subtraction of field elements. We first accumulate both the value to subtract and a multiple of the prime (that stems from reduction) by using multiply-accumulate instructions. Then we use subtract with borrow to simultaneously subtract both results. Note that this will result in remarkable speed differences between addition and subtraction. Addition could be implemented very efficiently by using the powerful multiplication engine. Subtraction is somewhat slower because SUBS and SBCS have to be used.

For addition and subtraction, we did use the powerful inline assembly capabilities of both, GCC and CLANG, that allowed us to avoid a significant amount of call overhead. In order to avoid operand fetches and stores wherever possible, we also made use of an inline assembly function that merges addition of a first operand with the curve-constant's multiple of a second operand ( $r := a + b * 121666$ ). Addition and subtraction of field elements follows the strategy from [DHH<sup>+</sup>15] by first processing the most significant word and then merging reduction of the two most significant bits and addition(subtraction) operation for the remaining seven words. We make use of the available carry bit 255 to obtain an implementation with only one single carry chain.

One additional optimization strategy was to bundle load and store operations together as much as possible in blocks. This way the pipeline latency on the M4 could be reduced. Isolated load and store operations account for two clock cycles each, while a sequence of  $n$  such operations only accounts for  $n + 1$  cycles.



**Table 2:** Sequence of executing the 64 partial products of words  $A_i \times B_j$  used for schoolbook multiplication of 256 bit operands.

	A0	A1	A2	A3	A4	A5	A6	A7
B0	1	5	10	15	20	25	28	48
B1	0	6	11	16	21	26	29	31
B2	2	7	12	17	22	27	30	32
B3	3	8	13	18	23	49	50	51
B4	4	9	14	19	24	52	53	54
B5	33	36	39	42	45	55	56	57
B6	34	37	40	43	46	58	59	60
B7	35	38	41	44	47	61	62	63

### 8.3.1 Field multiplication for ARM Cortex M4

Table 2 depicts the sequence (0...63) in which each of the 64 partial products of the schoolbook multiplication of the input operand words A0 ... A7 and B0 ... B7 is executed. Our optimization of the multiplication strategy does not seem to follow a regular pattern at first sight.

We use this sequence several reasons. Firstly, we observed that keeping as many input operands in registers as possible is equally important as avoiding stack spills of intermediate multiplication results. Secondly, it is worth noting that a multiplication actually costs *fewer* instructions if two intermediate results are to be accumulated at the same time. If only one intermediate result is to be accumulated, a MULAL instruction has to be used, which typically requires clearing of a scratch register (+1 instruction).

Basically four subblocks may be distinguished. Initially input operands B0 to B4 are cached in registers and multiplied one after the other with input operands A0 to A4. In the process of multiplication increasingly more registers were required for holding intermediate multiplication results. Completed result words that had been fully accumulated were spilled on the stack in order to free registers for more temporary results. Still starting with the multiplication with input A5, the input operand registers B3 and B4 were required as scratch registers for storing temporaries (multiplication steps 25 to 32). Ultimately also B0 had to be discarded. Then, in order to complete words 5 to 6 of the multiplication results (for freeing completed result words by writes to the stack), multiplication of input operands B5 to B7 with A0 to A4 is performed (33 to 47). Subsequently the multiplication result word 7 could be completed after the multiplication of A7 and B0 (Step 48). Finally the multiplications of input words A6 to A7 with B3 to B7 is calculated. Here the values A5 to A7 were cached in registers.

During the multiplication only 8 register spills were necessary for storing the lower-most result words temporarily on the stack. After multiplication the upper 8 result words of the 512 bit multiplication result were reduced within the register set before storing the reduced result back to memory.

### 8.3.2 Field squaring for ARM Cortex M4

For squarings we again make use of a special property of the Cortex-M4 instruction set which allows for two different types of carry chain. Either the ADDS and ADCS instructions may be employed (storing the carry bit in the status register) or UMLAL and UMAAL instructions (which do not modify the carry bit). The latter instructions store carries in full registers. We make use of this by using addition instructions for doubling the off-diagonal parts (with the exception of the product term  $A1 \cdot A0$ ), while we make use of integrated multiply-

**Table 3:** Sequence of executing the partial product words  $A_i \times A_j$  used for schoolbook squaring of 256 bit operands .

	A0	A1	A2	A3	A4	A5	A6	A7
A0	1	2						
A1	0	3						
A2	5	6	15					
A3	4	11	12	19				
A4	8	9	16	23	32			
A5	7	13	20	24	27	34		
A6	10	17	21	25	28	30	35	
A7	14	18	22	26	29	31	33	36

accumulate operations everywhere else.

In comparison to multiplication, we were able to hold more input operands in registers. The following table depicts the sequence (0..36) in which the partial products were calculated.

Throughout the calculation we distinguish between off-diagonal multiplication results (which require subsequent doubling) and diagonal multiplication results which were accumulated by use of multiplication instructions. Just as for multiplication, squaring is merged with reduction. This way only 5 register spills to the stack were required for storing intermediate multiplication results.

## 8.4 Implementation of the hash functions

For the calculation of SHA512 we use the optimized assembly code for the Cortex M0 architecture on both targets. This code fully unrolls the inner loop of the add-rotate-xor algorithm. We also make use of the special instructions for endianness-change. For the Cortex M4, some further speedup would be possible, when exploiting the availability of the "upper" registers. We did, however, stick with the more compact Cortex M0 code also because hashing was not the performance bottle neck.

## 9 Experimental results

In the following sections we will report on experimental results obtained from several different microcontroller targets, nRF51822, nRF52832, STM32F407, STM32F411 and STM32L476. We decided to include figures for all of these instead of selecting one particular chipset since in the course of our analysis, we observed that for the Cortex M4 architecture a major difficulty arises regarding speed benchmarking. Unlike for smaller architectures we observed that the highly target-specific performance of the flash memory subsystem plays a major role for the actual speed.

We did observe the most remarkable effect for the microcontroller STM32L476 targeting specifically ultra-low-power applications. Note that for ultra-low-power operation a suitable compromise between increased microcontroller speed and increased power consumption due to speculative flash accesses has to be found. We attribute our finding that the cycle counts for the analyzed primitives (depending on the power-consumption configuration) could increase by almost 40% when increasing the clock frequency from 16 MHz to 80 MHz mainly to such type of optimization. Obviously this makes fair speed benchmarking very difficult.

For the high-performance-family devices STM32F411 and STM32F407 from the same manufacturer, we observed that the influence of the clock frequency on performance is still present, but much smaller. Specifically for the STM32F411 almost no speed reduction was observed also at its highest clock frequency. In addition to flash timing issues, it is worth noting that for the STM32F407 device we observed some further dependence on the RAM memory configuration. This device disposes of so-called core-coupled memory (CCM). The timings reported here were obtained when placing the execution stack to CCM. Speed figures were observed to be significantly faster than when placing the stack in conventional RAM region.

As a result, we conclude that for speed benchmarking for cryptography implementations it is best to compare results obtained at lower clock frequencies. According to our results then some variations between different microcontroller suppliers still do exist, however the resulting values are at least of the same order of magnitude. We suggest, that the STM32F411 as a typical medium-size implementation for IIoT applications might be well suited as kind of reference platform for speed benchmarking.

## 9.1 Field arithmetic

In table 4 the speed results for the field arithmetic are summarized. Despite the mentioned difficulties regarding benchmarking, we come to the conclusion that our field arithmetic is significantly more efficient than the previously best published results on the Cortex M4 microcontroller in [FA17], specifically regarding multiplication and squaring of field elements.

The speedup obtained for the field arithmetic in comparison to reports from Aranha and Fujii in [FA17] in our opinion might stem from the following differences. Firstly for multiplication and squaring we did merge multiplication and squaring functions with reductions. This allowed us to hold more operands in registers. Secondly with the realized level of optimization regarding multiplication and squaring, the performance of addition and subtraction within the X25519 calculations starts becoming important as well. For these simpler operations call overhead becomes significant and use of inline assembly functions highly beneficial.

For the purpose of comparison, we also did add timings for the field  $\mathbb{F}_{(2^{127}-1)^2}$  as used in the construction FourQ in [CL15, LLP<sup>+</sup>17]. Note that our timings for  $\mathbb{F}_{(2^{255}-19)}$  are significantly faster despite the fact that the group order is comparable. As a result, we expect that a large fraction of the algorithmic speedup that is made possible by the endomorphisms of FourQ is lost by less efficient field arithmetic in [LLP<sup>+</sup>17]. Note, that the most relevant figure for the speed of Diffie-Hellman on FourQ is field multiplication, where the difference to our results is particularly large. We have reviewed the field operations in [LLP<sup>+</sup>17] and believe that significant further speedups should be possible also for FourQ. For field multiplication, for instance, we believe that a speedup in the range of 25 percent might be feasible in comparison to [LLP<sup>+</sup>17] when using the methods described in this paper.

## 9.2 X25519 Diffie-Hellman

In table 5 we summarize the results for the X25519 function for different microcontrollers and different clock frequencies. Our fastest result for X25519 on the M4 executes in as little as 609.779 cycles and is, thus, roughly 3 and 2.5 times faster than the reports in [dG15] (1816351) and [DSS16] (1563852) respectively and also significantly faster than the previously fastest result (907.240 cycles) from [FA17]. It is worth noting that in contrast to [FA17] we did use (in line with [HL17]) constant-time swaps of pointers instead of swapping full field elements. Note that for internal memories of Cortex M4 and M0 access timing is deterministic. When swapping pointers we expect both, more speed and less

**Table 4:** Field arithmetic on different targets at different frequencies  $f$  (/MHz). Columns  $*A_0$  ( $+ * A_0$ ) contain clock cycles for multiplication with the field constant  $A_0 = 121666$  and merged addition and multiplication  $x + y * A_0$ . Cycle count for the nRF51 target was obtained with the CLANG compiler with compile switch `-O2` while for the ST Microelectronics microcontroller we did use GCC 4.9.2 with optimization setting `-O2`.

Target	$f$	$x + y$	$x - y$	$*A_0$	$+ * A_0$	$x^2$	$x * y$	
nRF51822	16	120	147	193	-	998	1478	$\mathbb{F}_{(2^{255}-19)}$ , this work
STM32F411	?	73	77	129	-	563	631	$\mathbb{F}_{(2^{255}-19)}$ , [DSS16]
MK20DX	72	86	86	76	-	252	276	$\mathbb{F}_{(2^{255}-19)}$ , [FA17]
STM32F411	16	55	72	-	58	153	222	$\mathbb{F}_{(2^{255}-19)}$ , this work
STM32L476	16	52	65	-	55	153	220	$\mathbb{F}_{(2^{255}-19)}$ , this work
STM32L476	80	95	124	-	95	168	237	$\mathbb{F}_{(2^{255}-19)}$ , this work
nRF52832	64	62	70	-	65	162	229	$\mathbb{F}_{(2^{255}-19)}$ , this work
STM32F407	84	56	74	-	56	155	223	$\mathbb{F}_{(2^{255}-19)}$ , this work
STM32F407	84	86	-	-	-	215	358	$\mathbb{F}_{(2^{127}-1)^2}$ [LLP <sup>+</sup> 17]

side-channel leakage. Note however, that our implementation requires (unlike [FA17]) to be run with using internal RAM memory with constant access times. Our implementation optionally also allows for swapping field elements instead of pointers. According to our own measurements the penalty of doing so accounts roughly for additional 50.000 clock cycles.

Again we also have added speed benchmarks for FourQ from [LLP<sup>+</sup>17] for reference. Note that comparing of the fundamentally different algorithms X25519 and Diffie-Hellman on FourQ is difficult. For instance, when using the endomorphisms in FourQ quite large tables in RAM are required (required stack size is unfortunately not reported in [LLP<sup>+</sup>17]). Also note that the code size is about a factor of three larger than for our X25519 implementation. Despite the fact that our X25519 implementation (ca. 625,500 cycles) is much more adapted to small targets, our observed speed is very competitive in comparison to the reported result for Diffie-Hellman on FourQ (542,900 cycles including point decompression).

#### Update from July 2019 in comparison to the TCHES paper version:

The fact that X25519 could be implemented extremely efficiently on the ARM Cortex M4 is also illustrated by a recent work of Emil Lenngren in [Len18]. His X25519 implementation is claimed to execute in only 548,873 cycles. This result is particularly remarkable since this count already includes constant-time field element swaps by logic instructions (roughly worth 50k instructions). We see the main difference to our implementation in the fact that Emil Lenngren uses a custom function-call application binary interface (ABI). His implementation implements the full X25519 function in assembly and passes full field element results in registers. This technique allows him to avoid a significant amount of memory operations.

### 9.3 Partially augmented AuCPace25519

We have implemented AuCPace by using an asynchronous execution engine as suggested in [HL17]. Table 6 summarizes the speed results for individual substeps for the Cortex M0 (nRF51822) and different Cortex M4 microcontrollers.

We observed a speedup of roughly a factor of two in comparison to the results of [HL17] regarding the Elligator2 substep. In [HL17] the Elligator2 mapping algorithm was calculated by use of two separate exponentiations for inversion and calculation of

**Table 5:** Speed of X25519 scalar multiplication (FourQ) (on different targets, clock and memory configurations). The timings marked with  $(p)$  were obtained with enabled flash pre-fetch engines which increase current consumption.

Target	f / MHz	X25519	
nRF51822	16	3,474,201	this work
STM32F411	?	1,816,351	[dG15]
STM32F411	?	1,563,852	[DSS16]
MK20DX	72	907,240	[FA17]
STM32L476	16; 80 <sup>(p)</sup> ; 80	609,779; 857,002; 971,272	this work
nRF52832	64	634,567	this work
STM32F411	16; 100 <sup>(p)</sup> ; 100	625,347; 625,449; 734,554	this work
STM32F407	16; 84(p); 168 <sup>(p)</sup> ; 168	625,358; 626,719; 655,891; 847,048	this work
?	?	548,873	[Len18]
STM32F407	84 <sup>(p)</sup>	542,900 (FourQ)	[LLP <sup>+</sup> 17]

**Table 6:** Cycle counts for the nRF51822 Cortex M0 (STM32F411 Cortex M4) microcontrollers running at 16 MHz for SHA512, Elligator2, a X25519 Montgomery ladder step (LS), Inversion ( $1/x$ ) and a complete partially augmented AuCPace protocol run.

Target	SHA512	Elligator2	LS	$1/x$	AuCPace
nRF51822	21,564	289,276	13,521	258,291	7,345,820
STM32F411	21,130	46,032	3,163	42,590	1,351,381

the Legendre symbol  $\chi$ . In our work, we make use of the inverse square root algorithm for calculating Elligator2 with one single field exponentiation. In total this accounts for roughly 4 percent of a speedup regarding the balanced PACE (CPace) protocol runs on the Cortex M0.

Our results for the Cortex-M4 microcontroller family are faster by a factor of 5.4 in comparison to the Cortex M0, showing that this microcontroller architecture with its signal-processing instructions is by far better suited and likely also much more power-efficient for implementing complex asymmetric cryptography.

In table 7 the memory consumptions for the asynchronous execution object ACE from [HL17] and the stand-alone algorithm X25519 are summarized. The figures for the ACE object also include a Salsa20-20 based pseudo-random-number generator and the implementation for SHA512. For the Cortex M4 version, the total RAM requirement amounts to 532 bytes including static memory and stack. The stand-alone synchronous X25519 implementation (no state in static memory) for the Cortex M4 needs 444 bytes of stack memory and 3,324 bytes of flash and improves, thus, upon previous work [DSS16, FA17].

All of our code avoids secret-dependent branches and is, thus, executing in constant time on target-platforms with deterministic RAM memory access timings, such as typically found in ARM Cortex M0 and M4 microcontrollers.

## 10 Discussion and conclusion

In this paper we have presented a comprehensive analysis regarding possible optimizations for verifier-based password-authenticated key exchange for the setting of resource-constrained servers. Our analysis did cover all of protocol design, protocol security proof,

**Table 7:** Memory consumption in bytes for asynchronized implementation of AuCPace (ACE) and X25519 for Cortex M0 and M4 microcontrollers. Results were obtained with `arm-none-eabi-gcc -O2` (gcc version 4.9.3). RAM consumption is separated in static memory (stack memory) respectively.

Target	RAM	ROM	RAM	ROM	
Target	ACE	ACE	X25519	X25519	
Cortex-M0	264 (396)	11252	0 (572)	6108	this work
Cortex-M4	264 (268)	8896	0 (444)	3324	this work
Cortex-M4				4152	[FA17]
Cortex-M4				3786	[DSS16]

algorithmic optimization regarding group operations and field arithmetic and assembly-level fine-tunings.

Our construction allows for particular advantages in IIoT settings where a large number of small server nodes should be expected to operate with the same passwords, such as, e.g., the case in industrial plants. In addition to the conventional notion of verifier-based PAKE, our construction also allows for a partial augmentation operation mode that essentially halves the server’s computational complexity of the password verification step. Our protocol optionally allows also for pre-computation attack resistance.

Our construction with full augmentation imposes a complexity of four exponentiations in total on the server, one of which could be pre-computed prior to each login. The user-perceived login delay, thus is governed by the time consumed for calculating three scalar multiplications. The optional pre-computation attack resistance of strong AuCPace comes at a cost of one additional scalar multiplication for the server.

Our construction is two exponentiations faster for the server than all previously known verifier-based PAKE constructions when instantiated in its partially augmented variant. In the setting of [HL17] where one scalar multiplication accounts for about two seconds this results in a clearly perceivable usability gain in comparison to previously known protocols requiring at least three scalar multiplications.

Moreover our construction inherently allows for using strong memory-hard password hashing also on small servers since the costly memory-consuming operations are deferred to the clients.

The composability of the AuCPace security guarantees facilitates security analysis for use of AuCPace, e.g., as a building block in larger constructions, such as a centralized ticket-based user-credential distribution framework for industrial plants.

In contrast to most previous Diffie-Hellman based V-PAKE constructions with analysis in the UC framework, our security proof provides guarantees also in the stronger fully adaptive adversary model which allows for corruptions at any time during session establishment.

Finally, we have presented performance benchmarks of an instantiation targeting common microcontroller platforms coined AuCPace25519 which instantiates our protocol with using the primitives X25519, Elligator2 and SHA512.

The protocol runs in only 1,351,381 (7,345,820) cycles for a partially augmented protocol run on an ARM Cortex-M4 (M0) microcontroller respectively. On the M4 AuCPace requires only 8896 (532) bytes of flash (RAM) memory. There the X25519 Diffie-Hellman protocol sub-step executes in as little as 609,779 cycles. Our implementation, thus, did<sup>2</sup> set up a new speed records for both, (V)-PAKE protocols and X25519 Diffie-Hellman key exchange on this important embedded CPU architecture platform.

<sup>2</sup>Update July 2019 to the TCHES paper version: Now this speed record is held by Emil Lenngren’s implementation [Len18].

This illustrates also that on the Cortex M4 X25519 could be implemented very competitively, even in comparison to the best currently published implementations that exploit additional structure in elliptic curves, such as endomorphisms.

Summing up, we believe that all of the individual components presented in this paper in combination might yield a solution particularly tailored for the needs of real-world resource-constrained IIoT environments, such as notably intrinsically safe power limited industrial instrumentation.

## 11 Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments and advice and their great care with the manuscript. We also acknowledge inspiring discussions with Daniel Rausch, Ralf Küsters, Björn Tackmann, Julia Hesse, Denis Kügler, Marc Fischlin, Mike Hamburg and Peter Schwabe.

## References

- [ACCP08] Michel Abdalla, Dario Catalano, Céline Chevalier, and David Pointcheval. Efficient two-party password-based key exchange protocols in the UC framework. In *Topics in Cryptology - CT-RSA 2008, The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*, pages 335–351. 2008. 9, 16
- [AFP05] Michel Abdalla, Pierre-Alain Fouque, and David Pointcheval. Password-based authenticated key exchange in the three-party setting. In Serge Vaudenay, editor, *Public Key Cryptography - PKC 2005, 8th International Workshop on Theory and Practice in Public Key Cryptography, Les Diablerets, Switzerland, January 23-26, 2005, Proceedings*, volume 3386 of *Lecture Notes in Computer Science*, pages 65–84. Springer, 2005. 7
- [AP05] Michel Abdalla and David Pointcheval. Simple password-based encrypted key exchange protocols. In Alfred Menezes, editor, *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, volume 3376 of *Lecture Notes in Computer Science*, pages 191–208. Springer, 2005. 8
- [BDKJ16] Alex Biryukov, Daniel Dinu, Dmitry Khovratovich, and Simon Josefsson. The memory-hard argon2 password hash and proof-of-work function. Technical report, Internet-Draft draft-irtf-cfrg-argon2-00, Internet Engineering Task Force, 2016. Work in Progress, 2016. 3
- [BDL<sup>+</sup>11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2011. 37, 38
- [Ber06] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer-Verlag Berlin Heidelberg, 2006. <http://cr.yp.to/papers.html#curve25519>. 11



- [Ber14] Daniel J. Bernstein. 25519 naming. Posting to the CFRG mailing list, 2014. <https://www.ietf.org/mail-archive/web/cfrg/current/msg04996.html>. 11
- [BFK09] Jens Bender, Marc Fischlin, and Dennis Kügler. Security analysis of the PACE key-agreement protocol. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio Agostino Ardagna, editors, *Information Security, 12th International Conference, ISC 2009, Pisa, Italy, September 7-9, 2009. Proceedings*, volume 5735 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2009. 3, 5, 6, 8, 16, 35, 52
- [BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 967–980. ACM, 2013. 11, 16, 37, 38
- [BL19] Daniel J. Bernstein and Tanja Lange. SafeCurves: Choosing safe curves for elliptic-curve cryptography. Definition of Twist security. (accessed on 15 January 2019), 2019. <https://safecurves.cr.yt.to/twist.html>. 5
- [BLR04] Boaz Barak, Yehuda Lindell, and Tal Rabin. Protocol initialization for the framework of universal composability. *Cryptology ePrint Archive*, Report 2004/006, 2004. <https://eprint.iacr.org/2004/006>. 12
- [BM92] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. In *1992 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, May 4-6, 1992*, pages 72–84. IEEE Computer Society, 1992. 3, 6
- [BMP00] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using diffie-hellman. In Preneel [Pre00], pages 156–171. 3, 8
- [BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Preneel [Pre00], pages 139–155. 7
- [BRSS18] José Becerra, Peter Y. A. Ryan, Petra Sala, and Marjan Skrobot. An offline dictionary attack against zkpake protocol. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec 2018, Stockholm, Sweden, June 18-20, 2018*, pages 291–292, 2018. 6
- [Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. *IACR Cryptology ePrint Archive*, 2000:67, 2000. 8, 9
- [Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001. 7, 9
- [CGIP12] Jean-Sébastien Coron, Aline Gouget, Thomas Icart, and Pascal Paillier. Supplemental access control (PACE v2): Security analysis of PACE integrated mapping. In David Naccache, editor, *Cryptography and Security: From Theory to Applications - Essays Dedicated to Jean-Jacques Quisquater on the Occasion of His 65th Birthday*, volume 6805 of *Lecture Notes in Computer Science*, pages 207–232. Springer, 2012. 8

- [CHK<sup>+</sup>05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, pages 404–421, 2005. 3, 7, 10, 12, 15, 17
- [CKS08] David Cash, Eike Kiltz, and Victor Shoup. The twin diffie-hellman problem and applications. In Nigel P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 127–145. Springer, 2008. 32, 36
- [CL15] Craig Costello and Patrick Longa. Four<sub>11</sub>: Four-dimensional decompositions on a  $\mathbb{11}$ -curve over the mersenne prime. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 214–235. Springer, 2015. 42
- [CR03] Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 265–281. Springer, 2003. 5, 7, 9
- [dG15] Wouter de Groot. *A Performance Study of X25519 on Cortex-M3 and M4*. PhD thesis, Master thesis, Eindhoven University of Technology (Sep 2015), 2015. 42, 44
- [DHH<sup>+</sup>15] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. High-speed curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. *Des. Codes Cryptography*, 77(2-3):493–514, 2015. 37, 39
- [DSS16] Fabrizio De Santis and Georg Sigl. Towards side-channel protected X25519 on ARM Cortex-M4 processors. In *Proceedings of Software performance enhancement for encryption and decryption, and benchmarking, Utrecht, The Netherlands.*, 2016. 42, 43, 44, 45
- [EKSS09] John Engler, Chris Karlof, Elaine Shi, and Dawn Song. Is it too late for PAKE? In *Proceedings of the IEEE Web 2.0 Security and Privacy Workshop*, volume 5, page 17, 2009. 6
- [FA17] Hayato Fujii and Diego F Aranha. Curve25519 for the Cortex-M4 and beyond. *Progress in Cryptology-LATINCRYPT*, 2017. 42, 43, 44, 45
- [GMR06] Craig Gentry, Philip D. MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2006. 3, 5, 9, 10, 23, 24, 25, 33, 35, 36
- [Gre18] Matthew Green. Let’s talk about PAKE, 2018. <https://blog.cryptographyengineering.com/2018/10/19/lets-talk-about-pake/>. 3

- [Ham12] Mike Hamburg. Fast and compact elliptic-curve cryptography. *IACR Cryptology ePrint Archive*, 2012:309, 2012. 37
- [Ham17] Mike Hamburg. ISR trick for elligator2. Posting to the Curves mailing list, 2017. <https://moderncrypto.org/mail-archive/curves/2017/000939.html>. 37
- [HL17] Björn Haase and Benoît Labrique. Making password authenticated key exchange suitable for resource-constrained industrial control devices. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 346–364. Springer, 2017. 2, 4, 5, 6, 10, 39, 42, 43, 44, 45, 53
- [HL18] Björn Haase and Benoît Labrique. Repository for optimized X25519 field arithmetics code for ARM cortex M4 microcontrollers, 2018. <https://github.com/BjoernMHaase/fe25519>. 6
- [HR10] Feng Hao and Peter Ryan. J-PAKE: authenticated key exchange without PKI. *Trans. Computational Science*, 11:192–206, 2010. 6
- [HS14] Feng Hao and Siamak Fayyaz Shahandashti. The SPEKE protocol revisited. In Liqun Chen and Chris J. Mitchell, editors, *Security Standardisation Research - First International Conference, SSR 2014, London, UK, December 16-17, 2014. Proceedings*, volume 8893 of *Lecture Notes in Computer Science*, pages 26–38. Springer, 2014. 8
- [Jab96] David P. Jablon. Strong password-only authenticated key exchange. *Computer Communication Review*, 26(5):5–26, 1996. 3, 5, 6, 8
- [Jab97] David P. Jablon. Extended password key exchange protocols immune to dictionary attacks. In *6th Workshop on Enabling Technologies (WET-ICE '97), Infrastructure for Collaborative Enterprises, 18-20 June 1997, MIT, Cambridge, MA, USA, Proceedings*, pages 248–255, 1997. 3, 8
- [JKKX16] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). *Cryptology ePrint Archive*, Report 2016/144, 2016. <https://eprint.iacr.org/2016/144>. 55, 63
- [JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 456–486. Springer, 2018. 2, 4, 7, 12, 17, 24, 33, 36, 52, 54, 55, 62
- [KR17] Ralf Küsters and Daniel Rausch. A framework for universally composable diffie-hellman key exchange. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 881–900. IEEE Computer Society, 2017. 9
- [Kra05] Hugo Krawczyk. HMQV: A high-performance secure diffie-hellman protocol. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 546–566. Springer, 2005. 34, 63

- [Kra18] Hugo Krawczyk. The opaque asymmetric pake protocol, 2018. <https://tools.ietf.org/pdf/draft-krawczyk-cfrg-opaque-00.pdf>. 34
- [KTR13] Ralf Kuesters, Max Tuengerthal, and Daniel Rausch. The IITM model: a simple and expressive model for universal composability. Cryptology ePrint Archive, Report 2013/025, 2013. <https://eprint.iacr.org/2013/025>. 14
- [Len18] Emil Lenngren. Optimized X25519 for ARM cortex-M4 microcontrollers, 2018. <https://github.com/Emill/X25519-Cortex-M4>. 43, 44, 45
- [LL97] Chae Hoon Lim and Pil Joong Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 249–263. Springer, 1997. 10
- [LLM07] Brian A. LaMacchia, Kristin E. Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In Willy Susilo, Joseph K. Liu, and Yi Mu, editors, *Provable Security, First International Conference, ProvSec 2007, Wollongong, Australia, November 1-2, 2007, Proceedings*, volume 4784 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2007. 34
- [LLP<sup>+</sup>17] Zhe Liu, Patrick Longa, Geovandro C. C. F. Pereira, Oscar Reparaz, and Hwajeong Seo. Fourq on embedded devices with strong countermeasures against side-channel attacks. *IACR Cryptology ePrint Archive*, 2017:434, 2017. 42, 43, 44
- [LW15] Hanwook Lee and Dongho Won. Prevention of exponential equivalence in simple password exponential key exchange (SPEKE). *Symmetry*, 7(3):1587–1594, 2015. 8
- [Mac01] Philip MacKenzie. On the security of the SPEKE password-authenticated key exchange protocol. *IACR Cryptology ePrint Archive*, 2001:57, 2001. 8
- [MRA15] Karina Mochetti, Amanda C Davi Resende, and Diego F Aranha. zkpake: A simple augmented PAKE protocol. In *Proceedings of the Brazilian Symposium on Information and Computational Systems Security (SBSeq)*, 2015. 6
- [MSKD16] Mark Manulis, Douglas Stebila, Franziskus Kiefer, and Nick Denham. Secure modular password authentication for the web using channel bindings. *Int. J. Inf. Secur.*, 15(6):597–620, November 2016. 62
- [PJ12] Colin Percival and Simon Josefsson. The scrypt password-based key derivation function., 2012. <http://tools.ietf.org/html/josefsson-scrypt-kdf-00.txt>. 3, 10
- [Pre00] Bart Preneel, editor. *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*. Springer, 2000. 47
- [PW17] David Pointcheval and Guilin Wang. VTBPEKE: verifier-based two-basis password exponential key exchange. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 301–312. ACM, 2017. 2, 3, 7, 8, 12, 32, 52, 62

- [RS17] Joost Renes and Benjamin Smith. qdsa: Small and secure digital signatures with curve-based diffie-hellman key pairs. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, volume 10625 of *Lecture Notes in Computer Science*, pages 273–302. Springer, 2017. 11
- [SKI10] SeongHan Shin, Kazukuni Kobara, and Hideki Imai. Security proof of Aug-PAKE. *IACR Cryptology ePrint Archive*, 2010:334, 2010. 4, 12, 32
- [SOAA15] Stanislav Smyshlyaev, Igor B. Oshkin, Evgeniy K. Alekseev, and Liliya R. Ahmetzyanova. On the security of one password authenticated key exchange protocol. *IACR Cryptology ePrint Archive*, 2015:1237, 2015. 6, 32
- [Tac19] Björn Tackmann. Preliminary analysis of security of the apake candidates. Posting to the CFRG mailing list, 2019. <https://mailarchive.ietf.org/arch/msg/cfrg/euWBn5Nku0WFGKQ6qcITaCKVM-k>. 21
- [Wu98] Thomas D. Wu. The secure remote password protocol. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 1998, San Diego, California, USA*, pages 97–111. The Internet Society, 1998. 3, 6
- [Zha04] Muxiang Zhang. Analysis of the SPEKE password-authenticated key exchange protocol. *IEEE Communications Letters*, 8(1):63–65, 2004. 8

## A *Deprecated*: Patent circumvention strategy regarding Map2Point

Our construction shares with [JKX18] the requirement that an efficient hashing to group elements must be available for the elliptic curve's point group.

In previous versions of this paper, the authors assumed, that for some short Weierstrass-form curves patents might prevent use of this type of algorithm. This appendix was written in order to provide an alternative approach. After a more detailed patent analysis, the authors are now convinced that un-encumbered hash-to-curve options are available for all relevant curves. **We therefore no longer recommend the approach from this appendix for any real-world use.** We did keep it here only since it is referenced in the documentation and some discussions on the CFRG mailing list.

As an alternative to hash-to-curve, [PW17] has presented a construction TBPEKE based on two base points and an additional scalar multiplication (Pointcheval and Wang). Note that this construction is very similar to the balanced sub-protocol CPace presented in this paper.

In this appendix we deal with the question, whether a TBPEKE-style construction could also be used instead of CPace as a balanced sub-protocol component for AuCPace. I.e. the question is whether the TBPEKE construction could also be proven secure in the UC model. In our opinion, this answer could be given affirmatively. However, unfortunately, we see the need of further idealizing assumptions. Specifically, since the simulator needs to be able to derive the secret exponents of outputs of Map2Point calls, we see the need to use a version of a common reference string (CRS) model, where the simulator is given the power to provide a chosen string specifying the special points.

With this additional assumption, we come to the conclusion, that the balanced sub-step of TBPEKE could be proven secure also in the UC framework.

I.e. for avoiding hash-to-curve, one might consider to replace our technique for the calculation of the ephemeral generator  $G$  as  $G = \text{Map2Point}(H_1(PRS))$  by the TBPEKE equivalent of  $G = A \times C^{H_1(PRS)}$ , where as part of the proof  $A$  and  $C$  are provided as CRS to all parties. When giving the simulator the power to provide these points, it could define them such that the secret exponents are known. This gives the simulator access to the secret exponent of the generators, as needed by the CPace security proof strategy.

The essential property (as pointed out also in [BFK09]) required for secure use in the real world is that the discrete logarithm of  $G$  must be unknown for both, honest parties and the adversaries. Note that for any TBPEKE-based construction we see as important pre-requisite that the "nothing-upon-my-sleeve" problem related to the secrecy of the discrete log of the points  $C$  and  $A$  needs to be resolved in a trustworthy way.

Here we make the following suggestion. For any of the older short Weierstrass form elliptic curves we suggest to determine the curve points  $A$  and  $C$  by the following algorithm. For the point  $A$  ( $C$ ) we suggest to first take the packed little-endian encoding of the standardized curve's base point  $x$  ( $y$ ) coordinate and calculate  $\tilde{x}_A = \text{SHA512}(x)$  ( $\tilde{x}_C = \text{SHA512}(y)$ ). When doing so, there is a non-negligible probability that the  $x$ -coordinates  $\tilde{x}_A$  and  $\tilde{x}_C$  actually correspond to the  $x$ -coordinate of a point on the twist or possibly on a small subgroup. In this case we suggest to increment the  $x$  coordinate step by step by one until a point on the cryptographic group is returned. We then suggest to choose the one out of two  $y$ -coordinate candidates  $y_A$  and  $y_C$  such that the least-significant bit of the  $y$ -coordinate is zero.

Based on the assumption that no common mathematical structure is shared between the respective short Weierstrass curve and the Add-Rotate-XOR (ARX) algorithm SHA512, we conclude that it is justified to conjecture the secrecy of the discrete logarithms of  $A$  and  $C$ .



## B Notes regarding UC security of the PACE protocol variant from [HL17]

The protocol in [HL17] is closely related to the protocol CPace presented in this paper. This protocol and the specific optimization steps were not yet analyzed within the UC framework. The main difference to CPace stems from the strategy used for circumventing the patents on SPEKE.

While we do not detail a full UC security proof for this protocol, we nevertheless would like to sketch the necessary steps for executing it. In the notation of [HL17] the password  $pw$  corresponds to a password-based key  $\pi$  generated, e.g. by hashing the password. The difference of [HL17] to CPace essentially is that the calculation of the password related string  $PRS$  involves an additional symmetric encryption, not actually needed for securely implementing the protocol. In order to cover the patent circumvention protocol, we suggest to proceed as follows:

- Firstly we use the encrypted version of the messages  $s$  together with the nonce value and the message  $t$  that are exchanged at the beginning of the protocol for deriving the session id needed for the UC framework.
- We then use the symmetric Salsa20-20 primitive on the password in order to generate the XOR pad used in [HL17] and modify the definition of the password-related string  $PRS$  according to the patent circumvention construction  $s||t$ .
- In order to fend off relay attacks, it will be mandatory to incorporate identifiers for the parties (corresponding to the CI of CPace) into the input to the Map2Point function ( $H(s||t)$ ) such that not only the password is authenticated but also the client and server identities. This could e.g. be done by incorporating a channel identifier component into input parameter  $\pi$  (the password-derived key) used in [HL17].
- We then prove that the password dependent string ( $s||t$ ) generated this way matches iff the same password parameter  $\pi$  was used by both, server and client, ensuring that the password and the identities match. For this step, we essentially need the property that the entropy of  $\pi$  is preserved when extracting a random stream from  $\pi$  and nonce value by using Salsa20-20.
- The rest of the proof could then be executed by the same procedures as used in this paper. (Note that the explicit authentication step involving generation of several authenticators by one single run of SHA512 is not mandatory for securely implementing  $\mathcal{F}_{\text{pwKE}}$ .)



## C Strong AuCPace with pre-computation attack resistance

In [JKX18] the notion of pre-computation attack resistant or so-called "strong" verifier-based PAKE protocols has been introduced and formally defined by giving an ideal functionality  $\mathcal{F}_{\text{SaPAKE}}$  for the UC framework. In the same paper, Jarecki, Krawczyk and Xu also presented a generic compiler from any conventional UC-secure augmented PAKE protocol to a protocol providing pre-computation resistance.

Since AuCPace is proven secure in the UC framework, this "compiler" from [JKX18] could be employed for turning AuCPace into a pre-computation attack resistant "strong" PAKE protocol. For this approach, the salt value to use for the password hash is no longer openly communicated but calculated by the OPRF functionality  $\mathcal{F}_{\text{OPRF}}$ .

Unfortunately, and similarly to the considerations for the security proof of OPAQUE [JKX18] the generic "compiler" approach of [JKX18] would formally require a strict sequencing of the OPRF and the V-PAKE substeps. This would result in additional communication rounds.

When inserting the data exchange of the OPRF into the AuCPace messages, the composition theorems don't guarantee security as-is. Doing so requires further detail analysis. In this appendix, we show how a modification of the AuCPace protocol can be proven secure in the stronger  $\mathcal{F}_{\text{SaPAKE}}$  security model (figure 8 without the cyan part) without requiring the sequencing for the generic "compiler" approach.

Turning AuCPace into the pre-computation attack resistant "strong AuCPace" will essentially come at the cost of one additional exponentiation for the server and two for the client devices for each connection establishment.

### C.1 Differences between AuCPace and "strong AuCPace" protocols

The essential component for providing pre-computation resistance consists in keeping the salt value secret.

In order to do so, only minor adaptations are required, firstly for the password registration process and secondly for the AuCPace augmentation layer. The required modifications for password verification are shown in figure 12. The main difference from figure 3 is that the server no longer stores the salt value but only a secret exponent  $q$ .

Figure 13 displays the corresponding modifications for the session establishment for the augmentation layer sub-protocol from figure 4. The password file lookup on the server for a given user name returns the secret exponent  $q$  and the password verifier  $W$ . In case that the user database lookup fails (user unknown to the server), we suggest to continue with a random point  $W$  and hash the user name together with a server-specific secret  $SSS$  in order to obtain a username-specific secret exponent  $q = \text{H}(SSS || \text{username})$ . On the client side the point  $Z$  is blinded with a random value  $r$ . (The inversion operation  $1/(r(c_{\mathcal{J}})^2)$  for calculating the exponent used for un-blinding the point  $UQ$  is calculated modulo the order of the prime-order subgroup of the curve.)

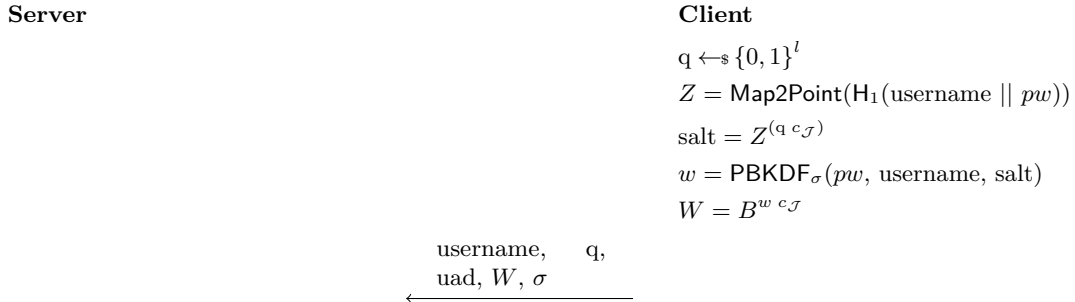
We would like to explicitly point the reader to the fact that for the proof, it was mandatory to explicitly consider the co-factors  $c_{\mathcal{J}}$ . It is also mandatory for the client and server to verify the points. (However the simplified verification strategies from section 3.4 also apply for the salt blinding substep in strong AuCPace.)

The additional computations correspond exactly to the required operations for the Diffie-Hellman OPRF function from [JKX18] when using the "compiler" approach.

### C.2 Sketch of the security proof for the Strong AuCPace protocol

We follow the same proof strategy as used for the augmented protocol, however for the proof we need the additional assumptions  $A1$  and  $A2$  from section 4.2 in order to be able to simulate the random oracle  $\text{H}_1$  according to the strategy from section 4.2.2. Additionally

## Store password operation for strong AuCPace



**Figure 12:** Modified protocol for password registering for strong AuCPace.

we need the "One-More" Diffie-Hellman assumption from [JKKX16], just as required for the generic "compiler" approach for the OPRF in [JKX18].

The first need for modifications in the proof shows up in game  $G_4$  of section 5.3.5. In this game the password is no longer accessible for the simulator. We need to provide the curve points  $U$  and  $UQ$  in a way indistinguishable from  $G_3$ , such that we could provide consistent data to the adversary in case of corruptions.

In  $G_4$  we simulate  $U$  as a random multiple  $B^{(u \cdot c_{\mathcal{J}})}$  of the group's base point and assign the server a scalar  $q$  for each user. Note that in order to be indistinguishable from  $G_3$ , the transmitted points need to come from the same distribution in both games. In order to guarantee this, we need the explicit co-factor treatment for generating the point  $U$  in the real world protocol. The server's response point  $UQ$  is calculated as  $U^{(q \cdot c_{\mathcal{J}})}$  as in the real-world protocol. We record the value of  $u$  for the event of corruptions.

If the client is corrupted, we learn the password and retrieve the secret exponent  $z$  for the point  $Z = B^z = \text{Map2Point}(H_1(\text{username}, pw))$  from our programmed value for the hash function  $H_1$ . We then return  $r = u/z$  to the adversary (same approach as within the proof for CPace). We then could calculate the salt value as  $Z^{(q \cdot c_{\mathcal{J}})}$ .

In case that the server is corrupted, we just return the value of  $q$  for the secret exponent of the OPRF.

We re-program the random-oracle for the PBKDF function such that the secret scalar  $w$  matches the password verifier  $W$  that might have been disclosed earlier to the environment by corrupting the server. The environment could distinguish  $G_4$  from  $G_3$  iff this re-programming operation fails, because the PBKDF hash has previously been queried for  $Z^{(q \cdot c_{\mathcal{J}})}$ . Since  $r$  is unknown to the environment and the group order of the point  $UQ$  has verified to be large, successful guessing of  $Z^{(q \cdot c_{\mathcal{J}})}$  happens with negligible probability.

Finally, it is necessary to show that an active adversary that impersonates the client is not able to calculate  $Z^{(q \cdot c_{\mathcal{J}})}$  offline even if she knows an arbitrarily large set of pairs  $(V, V^{(q \cdot c_{\mathcal{J}})})$  with  $Z \neq V$ . This case is ruled out when assuming the hardness of the "One-More" Diffie-Hellmann assumption from [JKKX16].

After compromising the server and learning  $q$ , however, the adversary could start with password hash queries and offline tests in both games.

This makes games  $G_4$  and  $G_3$  indistinguishable for the environment.  $\square$

StrongAuCPace augmentation layer

Server	Client
$x \leftarrow_{\$} \{1 \dots m_{\mathcal{J}}\}$	$r \leftarrow_{\$} \{1 \dots m_{\mathcal{J}}\}$
$X = B^{x \ c_{\mathcal{J}}}$	$Z = \text{Map2Point}(\text{H}_1(\text{username} \parallel pw)); U = Z^{r \ c_{\mathcal{J}}}$
	← username, $U$
$W, q = \text{lookup}W(\text{user})$	
$UQ = U^q \ c_{\mathcal{J}}$	
abort if UQ invalid	
	→ $\mathcal{J}, X, UQ, \sigma$
	$\text{salt} = UQ^{r \ (c_{\mathcal{J}})^2 \ c_{\mathcal{J}}}$
	abort if salt invalid
	$w = \text{PBKDF}_{\sigma}(pw, \text{user}, \text{salt})$
	abort if $X$ invalid
if lookup failed $PRS \leftarrow_{\$} \{0, 1\}^{k_2}$ ,	$PRS = X^w \ c_{\mathcal{J}}$
else $PRS = W^{x \ c_{\mathcal{J}}}$	

**Figure 13:** Strong AuCPace augmentation layer. In comparison to figure 4 in addition to the user name, the blinded point  $U$  is transmitted to the server. The server does reply with the point  $UQ$  instead of the clear-text salt value.

## D Requested documentation for the CFRG PAKE selection process

AuCPace and CPace have been nominated as candidate protocols for standardization. In 2019, the CFRG working group of the IETF has setup a corresponding process. In this section we will compile the requested information according to the requirements from RFC 8125.

- REQ1 "features regarding balanced/augmented versions":

AuCPace provides the features of an augmented PAKE, while CPace provides the features of a balanced PAKE protocol. Strong AuCPace provides the features of a pre-computation attack resistant augmented PAKE.

- REQ2 "proof, assumptions and models":

The security proof and the underlying assumptions for AuCPace and CPace are documented and published in the previous sections of this paper. The AuCPace paper has undergone scientific peer-review during the publishing process of the TCHES version.

A specific property is that CPace/AuCPace and strong AuCPace have been proven secure in the UC model for the functionalities  $\mathcal{F}_{\text{pwKE}}$ ,  $\mathcal{F}_{\text{apwKE}}$  and  $\mathcal{F}_{\text{SaPAKE}}$ . We consider it to be a somewhat exceptional fact that in contrast to other candidate protocols, we succeeded with carrying out the UC proofs in the stronger adaptive adversary model.

- REQ3 "protection against hostile environments":

As pointed out in the previous sections, AuCPace and CPace have been designed with consideration of the risk of implementation pitfalls from the very beginning. For the recommended reference instantiation AuCPace25519 and the incorporated primitives X25519, SHA512 and Elligator2 there is virtually no incentive of using non-constant time implementations, since variable-time algorithms don't allow for significant speedups. AuCPace has moreover been designed for using ephemeral values wherever possible in order to make side-channel attacks (SCA) more difficult (e.g. by restricting the attacker to one single trace for the ECC algorithms).

When considering use of AuCPace in hostile environments where invasive side-channel attacks need to be considered, we specifically recommend to use side-channel resistant implementations for the hash function  $H_1$  (e.g. masked implementations of SHA-3). Note that in this case intellectual property rights might become relevant and a patent-free implementation might not allow for some types of SCA countermeasures.

The reference implementation AuCPace25519 uses the hash function SHA512 for the real-world implementation of the random oracles. Due to the importance of the random-oracle idealization within in the proof, we considered imperfections of the hash function to be particularly critical for AuCPace. For this reason we have chosen a particularly high security margin (256 bit security level) for the hash. Moreover, we believe that using an Add-Rotate-Xor (ARX) construction might provide the best possible patent-free SCA mitigation strategy in hostile environments. In our opinion, the non-linear addition operation of ARX constructions might provide a more difficult SCA target than unprotected SBox-based or SPN-type constructions.

For highly hostile environments we recommend the implementation of AuCPace in SCA protected secure elements. AuCPace optimizes for this setting by allowing for particularly small password verifiers for the servers. (Note that on constrained cryptographic hardware, persistent tamper-protected storage might be a very limited resource).

- REQ4 "Use with ECC and mappings":

AuCPace and CPace are intended and designed for use with ECC. The requirements for the mapping algorithm are specified in section 4.2. A particular property is that the proof explicitly covers both, curves of prime order and curves with small co-factor.

When using AuCPace/CPace in conjunction with curves in short-Weierstrass form, we suggest to consider the patent circumvention approach from appendix A as a replacement for the mapping. This will allow for an un-encumbered implementation for the case that intellectual property rights have to be considered for constant-time mapping algorithms. Appendix A also clearly specifies the corresponding consequences with respect to the lowered security guarantees for this patent circumvention approach.

- REQ5 "Design choices": AuCPace/CPace is specifically designed for use in conjunction with constrained servers as explained in depth in section 3. We see the specific advantage of AuCPace in the fact that storage requirements for password verifiers is particularly small. We moreover believe that AuCPace features excellent power consumption figures, code-size and memory usage. We believe that this is an important aspect for real-world acceptance, specifically for use on secure-element hardware and constrained devices.
- REQ6 "application scenarios":

AuCPace is designed for use in conjunction with human-machine interfaces for authenticating individual human users. The most prominent use case might be a

password-based login on a web server which has not been integrated into a WEB-PKI infrastructure. We believe that this protocol might have its application also as a stand-alone scheme.

We see the main use-case for CPace as a replacement for pre-shared keys on interfaces that should be considered machine-to-machine interfaces. In this use-case we encourage the use of CPace without a user-identifier. Note that this allows for avoiding one communication round in the protocol. In this setting, we specifically encourage use of CPace for bootstrapping a PKI on constrained embedded devices based on a low-entropy shared secret. We believe that CPace might rather be suitable for integration into other protocols. Here, we believe that the composability guarantees of the UC security model might provide a significant advantage.

- REQ7 "privacy protections":

If user enumeration attacks are considered critical in a given application scenario, we recommend to use the strong AuCPace protocol from Appendix C.

- REQ8 "intellectual property rights":

We declare that we are not aware of any intellectual property rights restricting or limiting the use of AuCPace/CPace or subcomponents used within the protocol. Specifically we believe that the AuCPace reference implementation AuCPace25519 is not covered by any patent.

We are aware of third-party patents which possibly might cover some sub-aspects of efficient constant-time algorithms for mappings to ECC curves (E.g. inventors Coron and Icart). Unless detailed intellectual property right analysis has carried out, we recommend to use the circumvention approach from appendix A when using AuCPace/CPace in conjunction with prime-order curves in short-Weierstrass form.

In addition to the requirements above in the process of discussion on the CFRG list, the following additional questions were raised.

- REQ "agility":

AuCPace provides "crypto agility" by not relying on particular primitives and or parameters. Specifically it was specifically designed and proven secure also for security on non-prime order elliptic curves.

AuCPace moreover provides "patent circumvention agility" allowing for a flexible circumvention approach for the event that hashing to elliptic curve groups is covered by intellectual property rights.

AuCPace moreover provides "application agility" allowing for application-specific tailoring of the security guarantee vs. computational complexity balance. Specifically AuCPace allows for optional speedups using partial augmentation and for optional pre-computation attack resistance.

- REQ "A PAKE scheme should define how the explicit key confirmation is performed and clearly state whether this procedure is optional or mandatory.":

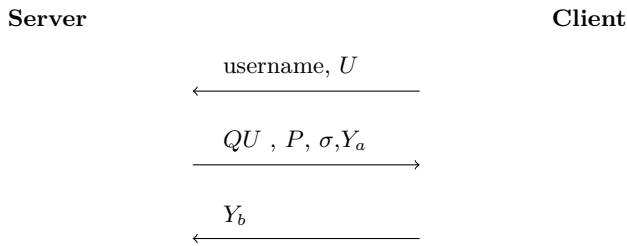
Both, AuCPace and CPace do not require explicit key confirmation but both do allow for explicit key confirmation. Sometimes authentication of some associated data related to a the channel is mandatory for an application and a key confirmation message might mainly serve for this purpose. (E.g. for fending off relay attacks.)

For AuCPace and CPace the implicit authentication of other associated data is possible by adding this information to the channel identifier CI string. For some applications, this feature might spare one communication round.

---

 Number of minimum protocol messages for Strong AuCPace
 

---



**Figure 14:** Minimum amount of required communication rounds, if the network layer is able to distinguish multiple concurrent sessions and the client starts.

- REQ "The authors should state if the two communicating parties can initiate the key exchange process at the same time, or it must always be the case that only one party can initiate the process.":

The situation is different for CPace and AuCPace. CPace is a perfectly symmetric construction. Both parties may take over the initiator or responder roles.

AuCPace is not symmetric. Either party may initiate the connection, however the lowest number of communication rounds is obtained if the client starts.

- REQ "The authors should clearly state the “round efficiency” of the PAKE protocol. In a standard two/multi-party secure computation setting, the “round” is defined as a step in which all parties can complete operations at the same time without depending on each other. In practice, a 2-round protocol could be implemented as 2 flows or 3 flows depending on the application context, but that’s more the implementation detail.":

For AuCPace it is somewhat difficult to answer this question due to the high amount of configurability, e.g. with respect to the optional explicit mutual authentication.

Similarly to other nominations with a proof in the UC model, the amount of required rounds also depends on the properties of the networking transport layer. In UC the session ID is used for distinguishing concurrent active sessions. If the networking layer (e.g. such as TCP) is able to distinguish multiple simultaneous sessions uniquely, there is no need for an additional round for establishing a session ID for addressing messages.

In this case, for the use-case of CPace with a pre-shared low-entropy secret on a machine-to-machine connection (with a single user and a single password), the protocol requires one message in each direction, i.e. two rounds, if implicit authentication is sufficient (as, e.g., in case of a secure channel protocol such as TLS).

In this setting, the augmented protocol AuCPace requires three rounds, as shown in figure 14.

Explicit mutual key confirmation adds up one additional message and explicit session ID generation again adds up one additional message for both AuCPace and CPace.

- REQ "If a PAKE scheme relies on the assumption of a trusted setup—more specifically, the discrete logarithm relationship between the two group elements in the system setup MUST be unknown—in anticipation of the worst but not impossible scenario, the authors should clearly state the security implications when the discrete logarithm relationship becomes known, and the subsequent mitigation measures.":

AuCPace has not been analyzed with respect to adversaries able to calculate the discrete logarithm ("quantum adversaries"). Anything we could give here in absence of a clear security model is somewhat hand-waving. Still we believe that the effect would not be a complete failure. One important helpful aspect is the fact that the generators are ephemeral.

Our reference implementation AuCPace25519 uses Elligator2 and does not have any discrete logarithm relationship issue. Also implementations using SWU or Icart's mapping on other curves will not suffer from this specific problem.

For this setting, we believe that the only thing an adversary would be capable of would be a "quantum dictionary attack" where a single offline password test involves one or two calculations of a discrete logarithm.

Regarding "quantum adversaries" a possible issue regarding known logarithm relationships shows up for implementations using the patent circumvention strategy from appendix A. The following paragraphs refer only to this patent-circumvention approach.

The main effect is that the discrete logarithm of the generator for CPace will be available to the adversary. Similarly the discrete logarithm of the point  $Z$  used for blinding the salt value in strong AuCPace will no longer be secret.

This will allow for attack strategies for active adversaries. Firstly, for strong AuCPace, pre-computation attack resistance will be affected, by allowing pre-computation of the password hash for an arbitrary number of passwords after one single online session. I.e. strong AuCPace will no longer be pre-computation attack resistant. Secondly, an active adversary will be able to mount offline attacks on the password.

As a mitigation strategy for the event of development of sufficiently powerful quantum computers, we suggest implementors to preferably use Montgomery or twisted Edwards curves, where Elligator2 is available. For implementations on curves such as P256 where integrated mappings such as SWU are available, we recommend to use SWU.

- REQ "A nomination SHOULD specify for which use-cases the protocol is recommended.":

CPace is mainly meant as replacement for pre-shared keys for interfaces that otherwise should be considered machine-to-machine interfaces. Specifically it is also considered for bootstrapping PKI infrastructure in settings where the initial trust anchor involves low-entropy secrets.

AuCPace is designed for constrained server devices and more powerful client devices providing the human-machine interface.

- REQ "We would like a scheme to explain how to avoid a user enumeration attack.":

In case that user enumeration attacks are considered feasible and critical, we suggest the Strong AuCPace protocol from Appendix C. Note that this comes at the expense of roughly 16 bytes longer password verifiers and communication bandwidth and an additional computational complexity of one scalar multiplication for the server and two scalar multiplication for the client.

- REQ "Information regarding computational complexity":

The computational complexity of the different AuCPace versions is summarized in table 1 in section 7.1.

- Question: "Is the considered security model relevant for all applications that PAKE is intended for?"



AuCPace has been designed for the use-case of critical industrial instrumentation where also functional safety aspects and the need for explosion protection make security mandatory. We believe that this highly sensitive application field will have higher demands than most other application domains that are considered for PAKE, for both, computational efficiency and security guarantees.

Specifically, we explicitly considered also aspects which become essential when the adversary is given physical access to server devices. This is one of the reasons, why we believe that using an augmented PAKE in combination with memory-hard password hashing is important. For this reason, AuCPace explicitly considers a password-hash work load parametrization  $\delta$  that may change over time, without requiring modification of the server implementation. Note that password hashing is referred to the client devices. For servers we recommend a rate-limiting that applies for repeated failures for online attacks.

We do not require and recommend any persistent data storage on client devices, which could help minimizing the impact of attacks.

We specifically encourage the use of AuCPace as a second security factor in addition to a first, possibly key-based or smart-card-based authentication mechanism.

- Question "Can it be integrated into IKEv2 / TLS?":

We believe that security-wise the best strategy would be to integrate both balanced PAKE and augmented PAKE into protocols such as TLS. We have made the experience that doing so will provide the best possible modularization of the security subcomponents by keeping the sensitive password handling outside of the main application code. We don't see any obstacle for integrating the balanced CPace protocol as a replacement for pre-shared key authentication into TLS 1.2 or TLS 1.3, specifically if CPace is operated without allowing for different accounts with several user names.

However integrating augmented PAKE into protocols such as TLS will create a significantly larger amount of software interfaces to the application layer. Reasonably the solutions for the use-cases of password change, re-authentication requests by the application would have to be dealt with on the TLS layer. We fear that this complexity might generate big obstacles.

For this reason, we make the following suggestion. We propose integration of the balanced CPace neatly with TLS. For the augmented PAKE use-case, CPace-TLS would be interfacing with a software module (V-PAKE handler). The V-PAKE handler would be providing both, the "AuCPace augmentation layer" which produces the password-related string *PRS* for a session and the interface used for re-authenticating users and implementing password-changes. Possibly the initial two protocol messages for PAKE session establishment could be implemented by use of a UDP-IP based service. Here sessions could be distinguished by ephemeral nonce-ids, which could serve for addressing and for cryptographic purposes, just mapping what is needed for universally composable models. Once the *PRS* has been established, the V-PAKE handler would invoke the TLS-CPace for establishment of the secure channel. The only important new API for TLS in this setting would be that the TLS-CPace layer should provide the V-PAKE handler an own session key derived from the TLS session key generated by CPace in addition to the one that it uses for TLS payload encryption and authentication.

The V-PAKE handler then could also be re-used on other protocols than TLS, for instance for the use-case of password-authenticated VPN access.

Possibly such a modularized approach (neat integration of CPace into the transport

layer, definition of a V-PAKE module and password/user API outside) might be more suitable when considering existing software architectures and infrastructure.

In our opinion the internal modular structure of AuCPace with a clear modularization into an augmentation layer and a subsequent balanced CPace substep allows for interesting options that might not be available for all of the PAKE nominations. Note that also in the paper Manulis, Stebila, Kiefer and Denham [MSKD16] modularized approaches for integration of PAKE into TLS have been suggested.

We would appreciate ideas and feedback with respect to this approach.

Concluding our statement, we would like to stress that our AuCPace / CPace proposal for CFRG should also be considered to be a compilation of excellent features from other CFRG PAKE nominations.

Specifically, in our opinion the pre-computation resistant strong AuCPace from Appendix C did merge the essence of OPAQUE from Jarecki, Krawczyk and Xu [JKX18] into AuCPace. The patent circumvention approach from Appendix A did integrate an important feature from VTBPEKE from Wang and Pointcheval [PW17] into CPace.

## E Overview over AuCPace protocol variants

AuCPace was designed as a highly configurable protocol family with respect to the following aspects

- *full* augmentation or *partial* augmentation
- implicit authentication or explicit mutual key confirmation
- protocols with ("strong AuCPace") and without (AuCPace) pre-computation attack resistance

This allows in total for 8 different combinations. Each of the corresponding real world protocols corresponds to a slightly different version of the ideal functionality  $\mathcal{F}_{\text{apwKE}}$  and cryptographic assumptions.

The security proofs in the main body of the paper cover the case of explicit mutual authentication without pre-computation attack resistance. At the same time, we also did want to cover the other alternative combinations of the feature sets as sketched above. This desire in addition to covering adaptive adversaries made the proofs more complex and more difficult to write down and read.

For this reason, we would like to summarize the differences regarding the assumption sets and ideal functionalities in this appendix. Each of the 3 different features corresponds to additions/omissions within the specification of the ideal functionalities.

### E.1 Differences between full and partial augmentation

Both variants share the same set of cryptographic assumptions.

In case of partial augmentation, the additional feature of  $\mathcal{F}_{\text{papwKE}}$  which allows for impersonation of the client to the server applies.

### E.2 Differences between explicit and implicit authentication

There is no difference with respect to the cryptographic assumptions for implicit and explicit authentication.

When a AuCPace protocol variant is used without explicit authenticator messages  $T_a$  and  $T_b$ , we need to give the adversary the power to issue `TestPwd` queries also if a session key has been delivered to one party. The sketch of the proof for implicit authentication is given in the main body of the paper.

For implicit authentication, security could be proven only for the *relaxed* versions of  $\mathcal{F}_{\text{apwKE}}$ . At the same time, the `TestAbort` queries will not be required for the proof and could be omitted. (In our opinion the changes in the functionalities with respect to `TestPwd` are related to the concepts of perfect and weak forward security as introduced in [Kra05].)

### E.3 Differences for versions with and without pre-computation attack resistance

In order to prove security for the "strong" protocols the additional "One-More" Diffie-Hellmann assumption as defined in [JKKX16] is required.

For proving security of pre-computation attack resistant "strong" variants of AuCPace, the functionalities of  $\mathcal{F}_{\text{apwKE}}$  omit the cyan parts in figure 8. The proof strategy sketch for the strong AuCPace variants is given in appendix C.