Security Analysis of CPace

Michel Abdalla[1], Björn Haase[2], and Julia Hesse*[3]

[1]CNRS and DI/ENS, PSL University
[2]Endress+Hauser Liquid Analysis
[3]IBM Research – Europe, Zurich

michel.abdalla@ens.fr, bjoern.haase@endress.com,
jhs@zurich.ibm.com

Abstract

In response to standardization requests regarding password-authenticated key exchange (PAKE) protocols, the IRTF working group CFRG has setup a PAKE selection process in 2019, which led to the selection of the CPace protocol in the balanced setting, in which parties share a common password. In subsequent standardization efforts, the CPace protocol further developed, yielding a protocol family whose actual security guarantees in practical settings are not well understood. In this paper, we provide a comprehensive security analysis of CPace in the universal composability framework. Our analysis is *realistic* in the sense that it captures adaptive corruptions and refrains from modeling CPace's Map2Pt function that maps field elements to curve points as an idealized function. In order to extend our proofs to different CPace variants optimized for specific elliptic-curve ecosystems, we employ a new approach which represents the assumptions required by the proof as libraries accessed by a simulator. By allowing for the modular replacement of assumptions used in the proof, this new approach avoids a repeated analysis of unchanged protocol parts and lets us efficiently analyze the security guarantees of all the different CPace variants. As a result of our analysis, all of the investigated practical CPace variants enjoy adaptive UC security.

1 Introduction

Security analysis and efficient implementation of cryptographic protocols are often split into separate working groups. As a result, subtle differences between the actually implemented and analyzed protocols easily emerge, for example when implementors slightly tweak the protocol to improve efficiency. An example where particularly aggressive optimizations for efficiency are implemented on the protocol level is CPace as specified in current internet drafts [26, 27]. CPace is a password-authenticated key exchange protocol (PAKE) [8], which allows two parties to establish a shared cryptographic key from matching passwords of potentially low entropy. PAKEs are extremely useful for establishing secure and authenticated communication channels between peers sharing

*Author supported by the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 786725 OLYMPUS.

short common knowledge. The common knowledge could be a PIN typed into different wearables in order to pair them, sensor readings recorded by several cars in order to create an authenticated platoon or a security code manually entered by an admin to connect her maintenance laptop with a backbone router.

On a high level, CPace works as follows. Given a cyclic group $\mathcal{G}$, parties first locally and deterministically compute a generator $g \leftarrow \mathsf{Gen}(\mathsf{pw}), g \in \mathcal{G}$ from their passwords in a secure way, so that g reveals as little information about the password as possible. Then, both parties perform a Diffie-Hellman key exchange by choosing secret exponents x and y, respectively, exchanging g^x and g^y and locally compute $K = (g^x)^y = (g^y)^x$. The final key is then computed as the hash of K together with session-identifying information such as transcript. The currently most efficient implementations of the above blueprint protocol use elliptic curve groups of either prime or composite order. To securely compute the generator, the password is first hashed to the finite field $\mathbb{F}_q$ over which the curve is constructed, and then mapped to the curve by a map called $\mathsf{Map2Pt}$. Depending on the choice of curve, efficiency tweaks such as simplified point verification on curves with twist security, or computation with only x-coordinates of points can be applied [25, 26]. Unfortunately, until today, it is not clear how these modifications impact security of CPace, and whether the protocol can be proven secure without assuming $(\mathsf{Map2Pt} \circ \mathsf{H})$ to be a truly random function.

A short history of CPace. In 1996, Jablon [34] introduced the SPEKE protocol, which performs a Diffie-Hellman key exchange with generators computed as $g \leftarrow \mathsf{H}_{\mathcal{G}}(\mathsf{pw})$, i.e. using a function $\mathsf{H}_{\mathcal{G}}$ hashing directly to the group. Many variants of SPEKE have emerged in the literature since then, including ones that fixed initial security issues of SPEKE. Among them, the PACE protocol [38, 9] aims at circumventing direct hashing onto the group with an interactive $\mathsf{Map2Pt}$ protocol to compute the password-dependent generators. From this, CPace [25] emerged by combining the best properties of PACE and SPEKE, namely computing the generator without interaction while avoiding the need to hash directly onto the group. More precisely, password-dependent generators are computed as $g \leftarrow \mathsf{Map2Pt}(\mathsf{H}(\mathsf{pw}))$. In 2020, the IRTF working group CFRG has chosen CPace as the recommended protocol for (symmetric) PAKE.

Prior work on the security of CPace. Bender et al. [9] conducted a game-based security analysis of the explicitly authenticated PACE protocol variants used in travel documents. Their work focusses on different variants of *interactive* $\mathsf{Map2Pt}$ constructions and hence does not allow for any conclusions about CPace which uses a (non-interactive) function $\mathsf{Map2Pt}$.

Static security of CPace, including function $\mathsf{Map2Pt}$ and some implementation artifacts such as cofactor clearing, was formally analyzed in [25]. Their work is the first to attempt a formalization of $\mathsf{Map2Pt}$ that allows for a security analysis. However, their proof was found to be insufficient by reviews done during the CFRG selection process [42, 31], and indeed, the claimed security under the plain computational Diffie-Hellman assumption seems to be difficult to achieve. Besides these issues, their work does not consider adaptive corruptions and implementation artifacts such as twist security or single-coordinate representations.

Abdalla et al. [2] analyzed static security of several EKE [8] and SPEKE variants in the UC framework, including SPAKE2 [5] and TBPEKE [39]. They indicate that their proof for TBPEKE could be extended to CPace with generators computed as $\mathsf{H}_{\mathcal{G}}(pw)$ (i.e., without function $\mathsf{Map2Pt}$) if the protocol transcript and password-dependent generator is included in the final key derivation hash. However, in practice it is desirable to avoid unnecessary hash inputs for efficiency reasons and protection against side-channel attacks.

In a concurrent work, Abdalla et al. [3] formalized the algebraic group model within the UC framework and proved that the SPAKE2 and CPace protocols are universally composable in the new model with respect to the standard functionality for password-based authenticated key exchange in [15]. Stebila and Eaton [19] provided a game-based analysis of CPace in the generic group model. As in [2], these further studies do not deal with adaptive security and only consider a basic version of CPace without Map2Pt and without considering any implementation artifacts.

The above analyses demonstrate that a basic version of CPace, which essentially is a Diffie-Hellman key exchange computed on hashed passwords instead of a public generator, is UC-secure if the attacker is restricted to static corruptions. Unfortunately, this leaves many open questions. Does this basic protocol remain (UC-)secure if we use generator $\mathsf{Map2Pt}(\mathsf{H}(pw))$ instead, as it is done in practice to avoid direct hashing onto elliptic curves? Can the protocol handle adaptive corruptions? Which impact on security do implementation artifacts have, such as co-factor clearing on a composite-order curve group, or single-coordinate representation as used in, e.g., TLS1.3? Can we reduce hash inputs in order to make the protocol less prone to side-channel attacks? Altogether, it turns out the security of the *actually implemented* CPace protocol is not well understood.

Our Contributions. In this paper, we provide the first comprehensive security analysis of the CPace protocol that applies also to variants of CPace optimized for usage with state-of-the-art elliptic curves. We identify the core properties of the deterministic Map2Pt function that allow to prove strong security properties of CPace. Crucially, we restrict the use of random oracles to hash functions only and refrain from modeling Map2Pt as an idealized function, as it would not be clear how to instantiate it in practice. We show that, using some weak invertibility properties of Map2Pt that we demonstrate to hold for candidate implementations, CPace can be proven secure under standard Diffie-Hellman-type assumptions in the random-oracle model and with only minimal session-identifiying information included in the final key derivation hash. Our security proof captures adaptive corruptions and weak forward secrecy[1] and is carried out in the Universal Composability (UC) framework, which is today's standard when analyzing security of password-based protocols. Our work provides the first evidence that SPEKE-type protocols can handle adaptive corruptions.

We then turn our attention to modifications of CPace and, for each modification individually, state under which assumptions the security properties are preserved. In more detail, our analysis captures the following modifications.

- Using groups of composite order $c \cdot p$, where p is a large prime and c is a small cofactor coprime to p.
- Realize Gen(pw) generator calculations using Map2Pt with either map-twice-and-add strategy or as single execution.
- Using single-coordinate-only representations of elliptic-curve points in order to speed up and facilitate implementation.
- Avoiding computationally costly point verification on curves with secure quadratic twists such as Curve25519 [10].

To demonstrate the security of these variants, we take a new approach that saves us from a repeated analysis of unchanged parts of CPace. Namely, we implement the CDH-type cryptographic assumptions required by CPace as libraries which a simulator can access. This allows for modular replacement of assumptions required in the security proof, and lets us efficiently analyze all the

[1] In the case of PAKE, weak forward secrecy is implied by UC security and hence achieved also by prior work. If key confirmation is added, then this gives a protocol with perfect forward secrecy as noted in [2].

different CPace variants' security guarantees. We believe that this new proof technique might be of independent interest in particular for machine-assisted proving, since reductions are captured in code instead of textual descriptions only.

As a side contribution, we identify a common shortcoming in all UC PAKE security definitions in the literature [15, 35, 32, 2], which impacts the suitability of these definitions as building blocks in higher-level applications. Namely, all these definitions allow a malicious party to learn the shared key computed by an honest party *without knowing her password*. We strengthen the definition to prevent such attacks, and demonstrate with our analysis of CPace that our fix yields a security definition that is still met by PAKE protocols.

In conclusion, our results demonstrate that CPace enjoys strong provable security guarantees in a realistic setting, and this holds for all its variants that have been proposed in the different elliptic-curve ecosystems.

1.1 Technical overview of our results

Map2Pt's impact on security. At its core, the CPace protocol is a SPEKE-type protocol, meaning that it is simply a Diffie-Hellman key exchange (DHKE) computed with a generator that each party *individually* computes from her password. Intuitively, the most secure choice is to compute $g \leftarrow \mathsf{H}_{\mathcal{G}}(pw)$, and indeed this was proven secure [2, 3] conditioned on H being a perfect hash function (or, put differently, a random oracle (RO)). However, DHKE-type protocols are most efficient when implemented on elliptic-curve groups, and it is not known how to efficiently hash directly onto such groups. Recent standardization efforts by the CFRG [21] show that, in practice, one would always first hash to the finite field $\mathbb{F}_q$ over which the curve is constructed, and then map the field element to the curve $\mathcal{G}$ using some curve-specific mapping $\mathsf{Map2Pt} : \mathbb{F}_q \to \mathcal{G}$. Hence, the generator in CPace can be assumed to be computed as $g \leftarrow \mathsf{Map2Pt}(\mathsf{H}(pw))$ for a H being a hash function such as SHA-3.

In order to analyze how the function $\mathsf{Map2Pt}$ impacts CPace's security, it is obviously not helpful to abstract $\mathsf{Map2Pt} \circ \mathsf{H}$ as a truly random function. In a first attempt to analyze under which properties of $\mathsf{Map2Pt}$ CPace remains secure, Haase et al. [25] assumed $\mathsf{Map2Pt}$ to be a bijection. Intuitively, a bijective $\mathsf{Map2Pt}$ function does not "disturb" the "nice" distribution of the prepended hash function, and in particular does not introduce any collisions. Besides the known shortcomings in their conducted analysis (the claimed security under CDH does not seem to hold, and their proof lacks an indistinguishability argument [42, 31]), it does not cover non-bijective mappings on widely used short-weierstrass curves such as NIST P-256. Hence, in our work we refrain from assuming $\mathsf{Map2Pt}$ to be a bijection. Instead, we introduce a property of *probabilistic invertibility*, which demands that, given an element g in the group $\mathcal{G}$, we can efficiently compute all preimages $h \in \mathbb{F}_q$ such that $\mathsf{Map2Pt}(h) = g$. On a high level, this invertibility property will aid the simulation of CPace since it allows to "tightly" link a group element g to a previously computed hash h and thus recognize collisions efficiently. Here, tightly/efficiently means without iterating over all hash queries in the system. We demonstrate that all mappings used in practice [21] are probabilistically invertible. As a result, we conclude that CPace implemented with current mappings enjoys strong security guarantees.

Adaptive security. Just like any other PAKE protocol, CPace comes with a large likelihood for idling. Indeed, in practice it will most likely be the same person who jumps between the two devices running the PAKE, to manually enter the same password, PIN or code. This gives room for

attackers to corrupt devices *during the run of the protocol*, and hence calls for analyzing security of CPace in the presence of *adaptive corruptions*. To our knowledge, there is no proof of adaptive security for any SPEKE-type protocol in the literature. In this work, we closely investigate CPace's guarantees under adaptive corruptions and come to an indeed surprising conclusion:

CPace enjoys adaptive UC security under the same DH-type assumptions that seem required for static security.

The challenge of proving adaptive security lies in the need to reveal suitable secret values computed by a previously honest party during the run of the protocol. For CPace, these are the secret Diffie-Hellman exponents x, y randomly chosen by parties. A bit simplified, our idea is to start the simulation of an honest party with g^z for a generator g of group $\mathcal{G}$ and randomly chosen exponent z, and hence independent of the actual (unknown) password used by that party. Upon corruption, the simulator learns pw and looks up the corresponding hash value $g^r = \mathsf{H}(pw)$ for which it knows r^{-1} thanks to H being modeled as a random oracle. This allows the simulator to compute the "actual" secret exponent $y \leftarrow zr^{-1}$ that the simulated party would have used if started with actual password pw. Crucially, no additional assumptions or secure erasures are required and, as we demonstrate in the body of our paper, this simplified strategy still works when generators are computed using $\mathsf{Map2Pt} \circ \mathsf{H}$. Altogether, our analysis shows that CPace enjoys UC-security under adaptive corruptions.

Falsifiable assumptions and a new approach to simulation-based proofs. A falsifiable assumption can be modeled as an interactive game between an *effcient* challenger and an adversary, at the conclusion of which the challenger can efficiently decide whether the adversary won the game [23]. Most standard cryptographic assumptions such as CDH, DDH, RSA, and LWE are falsifiable. An example of a non-falsifiable assumption is the gap simultaneous Diffie-Hellman assumption, which was used in prior CPace security analyses [2, 3] and features a full DDH oracle that cannot be efficiently implemented by the challenger. Intuitively, the DDH oracle seems inherent for proving UC security of CPace since the attacker (more detailed, the distinguishing environment) determines passwords pw used by honest parties and also receives their outputs, which is the final key K. More detailed, the attacker can deterministically compute the generator G used by an honest party from only pw, and it also receives the honest party's message g^x. The attacker can now enforce the final key to be a DDH tuple $K = g^{xy}$ by simply sending g^y to the honest party (we omit the final key derivation hash in this explanation for simplicity). Hence, to correctly simulate the final key output by an honest party under attack, the simulator relies on a DDH oracle. However, we observe that this oracle can be *limited to specific inputs* g, g^x that the attacker cannot influence. This turns out to be an important limitation, because the *restricted* DDH oracle $DDH(g, g^x, \cdot, \cdot)$ can actually be implemented efficiently using knowledge of trapdoor exponents r, r^{-1} of g. Thus, our conclusion is that CPace's security holds under falsifiable DH-type assumptions.

As another contribution, we define falsifiable assumptions as efficiently implementable libraries that a simulator can call. The advantage of this approach is that reductions to the underlying assumptions are *integrated* in the simulator's code, which will hence abort and detect itself whenever a query to the library solves the underlying hard problem. This makes reduction strategies readable from simulator codes and hence opens a new path for automatic verification of simulation-based proofs. While we demonstrate this only to work for proofs conducted in the UC framework and when using variants of strong CDH, we conjecture that our approach can be used for simulation-based proofs in arbitrary frameworks whenever only falsifiable assumptions are used.

Minimal protocol design. For optimal protection against side-channel attacks, we would like to have parties touch their passwords as little as possible. Optimally, passwords are only used to compute the generator of the DHKE. Unfortunately, in simulation-based frameworks a security proof often crucially relies on hashing of secrets, and indeed previous CPace security analysis has relied on the password being included in the final key derivation hash [2]. In this work we ask what the minimal set of protocol-related values is that needs to be included in both hash functions used in CPace. Perhaps surprisingly, we find that CPace's security can be proven when (1) the password hash does not get any additional inputs and (2) the final key derivation hash is over session-specific values and the Diffie-Hellman key. Regarding (1), we observe that the simulation strategy (described above for adaptive corruptions) works even if the generator g chosen by the simulator is used to simulate multiple instances of CPace, and where different parties use the same password: Choosing fresh secret exponents z_A for each such simulated party A ensures that all the revealed exponents $z_A r^{-}1$ are still uniformly distributed. Regarding (2), our simulation simply does not need to learn the password from an adversarial key derivation hash query: The simulator simply reads the simulated parts g^z and adversarial part Y of the transcript from the hash query and checks consistency of the query's format by checking whether it is a DDH tuple with respect to each trapdoor generated upon password hashing. Since there can be only a polynomial number of such queries, this simulation strategy is tight and efficient and saves us from hashing the password another time.

Implementation artifacts. Depending on the type of curve CPace is deployed in, the implementation will vary in certain aspects for which it is not clear how they will impact CPace's security. By adopting the security analysis to capture actual Map2Pt mappings used in practice we already demonstrated how to deal with the probably most important such artifact above. Closely related to this, we also analyze security of CPace when implemented on curves of composite order $p \cdot c$ with a small co-factor c, which needs to be "cleared" in order to ensure that parties use generators of the large subgroup. We can integrate this modification by chaining Map2Pt with a co-factor clearing function and by demonstrating that the resulting mapping is still probabilistically invertible. Technically, we "lift" our proof of security w.r.t simple Map2Pt described above by letting the simulator call a co-factor clearing class that ensures that simulated values will remain in the large subgroup.

A typical implementation pitfall is incorrectly implemented group-membership verification. As such a failure easily remains unnoticed, optimized resilient protocols such as X25519 and X448 [36] have been suggested for the conventional Diffie-Hellman use-case. We believe that we are the first to formalize the exact hardness assumption, the twist CDH problem sTCDH, under which the claimed resilience regarding group membership omission is actually justified. We show that under the sTCDH assumption, resilience with respect to incorrectly implemented point verification can also be achieved for CPace, when instantiated using single-coordinate Montgomery ladders on so-called "twist-secure" [12] elliptic curves. For details on how to deal with other implementation artifacts we refer the reader to Section 6 in the main body of the paper.

Roadmap. We introduce the PAKE security model in Section 2 and hardness assumptions and requirements for Map2Pt in Section 3. Details of the CPace protocol are in Section 4. Then we analyse CPace, first using a simplified CPace in Section 5 (modeling the map as random-oracle) and then extending the analysis to real-world instantiations using actually deployed mapping constructions, composite-order groups, details on twist security and single-coordinate representations in

Section 6. Full proofs (Appendix A), a description of issues with previous UC PAKE functionalities (Appendix C) and implementation recommendations (Appendix G) can be found in the appendices.

2 PAKE Security Model

We use the Universal Composability (UC) framework of Canetti [14] to formulate security properties of CPace. For PAKE, usage of the simulation-based UC framework comes with several advantages over the game-based model for PAKE introduced by Bellare et al. [7]. Most importantly, UC secure PAKE protocols preserve their security properties in the presence of adversarially-chosen passwords and when composed with arbitrary other protocols. Originally introduced by Canetti et al. [15], the ideal functionality $\mathcal{F}_{\mathsf{pwKE}}$ for PAKE (depicted in Fig. 1) is accessed by two parties, $\mathcal{P}$ and $\mathcal{P}'$, who both provide their passwords. $\mathcal{F}_{\mathsf{pwKE}}$ then provides both parties with a uniformly random session key if passwords match, and with individual random keys if passwords mismatch. Since an adversary can always engage in a session and guess the counterpart's password with non-negligible probability, $\mathcal{F}_{\mathsf{pwKE}}$ must include an adversarial interface TestPwd for such guesses. Crucially, only one guess against every honest party is allowed, modeling the fact that password guessing is an online attack and cannot be used to brute-force the password from a protocol's transcript. We refer the reader to [15] for a more comprehensive introduction to the PAKE functionality.

An ideal functionality for the SPEKE protocol family. Unfortunately, $\mathcal{F}_{\mathsf{pwKE}}$ is not suitable to analyze SPEKE-like PAKE protocols such as CPace, where session keys are computed as hashes of Diffie-Hellman keys (and possibly parts of the transcript). The reason is that $\mathcal{F}_{\mathsf{pwKE}}$'s TestPwd interface allows password guesses only *during* a protocol run, which requires a simulator to extract password guesses from the protocol's transcript. When the final output is a hash, the adversary might postpone its computation, keeping information from the simulator that is required for password extraction. To circumvent these issues, recently a "lazy-extraction PAKE" functionality $\mathcal{F}_{\mathsf{lePAKE}}$ was proposed and shown useful in the analysis of SPEKE-like protocols by Abdalla et al. [2]. $\mathcal{F}_{\mathsf{lePAKE}}$, which we also depict in Fig. 1, allows *either* one online *or* one offline password guess after the key exchange was finished. One might argue that usage of keys obtained from $\mathcal{F}_{\mathsf{lePAKE}}$ is never safe, since the adversary might eventually extract the key from it at any later point in time. This however can be easily prevented by adding a key confirmation round, which keeps an adversary from postponing the final hash query and guarantees perfect forward secrecy [2]. We refer the reader to [2] for a thorough discussion of $\mathcal{F}_{\mathsf{lePAKE}}$.

Our adjustments to $\mathcal{F}_{\mathsf{lePAKE}}$. The main difference between our $\mathcal{F}_{\mathsf{lePAKE}}$ and all PAKE functionalities from the literature [15, 35, 32, 2] is that we remove a shortcoming that rendered these functionalities essentially useless as building blocks for higher-level applications. More detailed, we remove the ability of the adversary to determine an honest party's output key in a corrupted session. The change can be seen in Fig. 1, where the dashed box shows the weakening that we simply omit in our version of $\mathcal{F}_{\mathsf{lePAKE}}$. In reality, nobody would want to use a PAKE where an adversary can learn (even set) the key of an honest party *without knowing the honest party's password*. This is not what one would expect from an authenticated key exchange protocol. In Appendix C we explain why existing PAKE protocols can still be considered secure, but also provide an illustrating example how this shortcoming hinders usage of PAKE functionalities in modular protocol analysis. In this paper, we demonstrate that CPace can be proven to protect against such attacks.

We also make two minor adjustments, which are merely to ease presentation in this paper. Namely, we omit roles since we analyze a protocol where there is no dedicated initiator, and we add an explicit interface for adaptive corruptions. The latter interface can only be asked upon getting instructed to do so by the environment.

How many keys can a PAKE functionality exchange? All PAKE functionalities in Figure 1 produce only a single key for a single pair of parties $\mathcal{P}, \mathcal{P}'$. This can be seen from the NewSession interface, which takes action only upon the first such query (from any party $\mathcal{P}$) and the corresponding second query by the indicated counterparty $\mathcal{P}'$. The motivation behind this design choice is simplicity in the security analysis: one can prove security of a PAKE protocol for only a single session, and then run arbitrary many copies of the PAKE functionality to exchange arbitrarily many keys (between arbitrary parties). Consequently, by the UC composition theorem, replacing all those copies with the PAKE protocol that provably realizes the single-session $\mathcal{F}_{\mathsf{pwKE}}$ is at least as secure.

On party authentication. The PAKE functionalities from Figure 1 require the party identifier of the counter party as input. In case of a mismatch (e.g., Alice wants to exchange a key with Bob, but Bob wants to exchange with Charlie), in an unattacked session, the functionalities' NewKey interface ensures that both parties obtain random keys. In case of the adversary playing a man in the middle, if he guesses both passwords correctly, the adversary can make the two parties output matching keys regardless of the parties' intentions who they want to exchange a key with.

3 Preliminaries

3.1 Notation

With $\leftarrow_{\mathrm{R}}$ we denote uniformly random sampling from a set. With $\mathsf{oc}(X, Y)$ we denote ordered concatenation, i.e., $\mathsf{oc}(X, Y) = X||Y$ if $X \leq Y$ and $\mathsf{oc}(X, Y) = Y||X$ otherwise. We use multiplicative notation for the group operation in a group $\mathcal{G}$ and hence write, e.g., $g \cdot g = g^2$ for an element $g \in \mathcal{G}$. $I_{\mathcal{G}}$ denotes the neutral element in $\mathcal{G}$. To enhance readability, we sometimes break with the convention of denoting group elements with small letters and write $X := g^x$. We denote by $\mathcal{G}_m$ a subgroup of $\mathcal{G}$ of order m, and with $\bar{\mathcal{G}}$ we denote the quadratic twist of elliptic curve group $\mathcal{G}$. Throughout the paper, we use λ as security parameter[2].

3.2 Cryptographic assumptions

The security of CPace is based on the hardness of a combination of strong and simultaneous Diffie-Hellman problems. To ease access to the assumptions, we state them with increasing complexity.

Definition 3.1 (Strong CDH problem (sCDH) [4])**.** Let $\mathcal{G}$ be a cyclic group with a generator g and $(X = g^x, Y = g^y)$ sampled uniformly from $(\mathcal{G} \setminus \{I_{\mathcal{G}}\})^2$. Given access to oracles $\mathsf{DDH}(g, X, \cdot, \cdot)$ and $\mathsf{DDH}(g, Y, \cdot, \cdot)$, provide K such that $K = g^{xy}$.

We note that sCDH is a weaker variant of the so-called gap-CDH assumption, where the adversary has access to "full" DDH oracles with no fixed inputs. Next we provide a stronger variant of sCDH where two CDH instances need to be solved that involve a common, adversarially chosen element.

[2]For the hardness assumptions on elliptic curve groups, e.g. for the sCDH and sSDH problems, where security depends on the group type and the group order p, the bit size of p implicitly serves also as a further security parameter.

Session initiation

On $(\mathsf{NewSession}, sid, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ from $\mathcal{P}$, send $(\mathsf{NewSession}, sid, \mathcal{P}, \mathcal{P}')$ to $\mathcal{A}$. In addition, if this is the first NewSession query, or if this is the second NewSession query and there is not yet a record $(sid, \mathcal{P}, *, *)$, then record $(sid, \mathcal{P}, \mathcal{P}', \mathsf{pw})$ and mark this record fresh.

Active attack

- On $(\mathsf{TestPwd}, sid, \mathcal{P}, \mathsf{pw}^*)$ from $\mathcal{A}$, if $\exists$ a fresh record $\langle sid, \mathcal{P}, \mathcal{P}', \mathsf{pw}, \cdot\rangle$ then:
  - If $\mathsf{pw}^* = \mathsf{pw}$ then mark it compromised and return "correct guess";
  - If $\mathsf{pw}^* \neq \mathsf{pw}$ then mark it interrupted and return "wrong guess".
- On $(\mathsf{RegisterTest}, sid, \mathcal{P})$ from $\mathcal{A}$, if $\exists$ a fresh record $\langle sid, \mathcal{P}, \mathcal{P}', \cdot\rangle$ then mark it interrupted and flag it tested.
- On $(\mathsf{LateTestPwd}, sid, \mathcal{P}, \mathsf{pw}^*)$ from $\mathcal{A}$, if $\exists$ a record $\langle sid, \mathcal{P}, \mathcal{P}', \mathsf{pw}, K\rangle$ marked completed with flag tested then remove this flag and do:
  - If $\mathsf{pw}^* = \mathsf{pw}$ then return $\boxed{K}$ to $\mathcal{A}$;
  - If $\mathsf{pw}^* \neq \mathsf{pw}$ then return $\boxed{K^{\$} \leftarrow_{\mathrm{R}} \{0,1\}^{\lambda}}$ to $\mathcal{A}$.

Key generation

On $(\mathsf{NewKey}, sid, \mathcal{P}, K^*)$ from $\mathcal{A}$, if $\exists$ a record $\langle sid, \mathcal{P}, \mathcal{P}', \mathsf{pw}\rangle$ not marked completed then do:

- If the record is compromised, or either $\mathcal{P}$ or $\mathcal{P}'$ is corrupted, then $K := K^*$.
- If the record is fresh and $\exists$ a completed record $\langle sid, \mathcal{P}', \mathcal{P}, \mathsf{pw}, K'\rangle$ that was fresh when $\mathcal{P}'$ output (sid, K'), then set $K := K'$.
- In all other cases pick $K \leftarrow_{\mathrm{R}} \{0,1\}^{\lambda}$.

Finally, append K to record $\langle sid, \mathcal{P}, \mathcal{P}', \mathsf{pw}\rangle$, mark it completed, and output (sid, K) to $\mathcal{P}$.

Adaptive corruption

On $(\mathsf{AdaptiveCorruption}, sid, \mathcal{P})$ from $\mathcal{A}$, if $\exists$ a record $\langle sid, \mathcal{P}, \mathcal{P}', \mathsf{pw}\rangle$ then:

- if (sid, K) was output to $\mathcal{P}$, send $(sid, \mathcal{P}', \mathsf{pw}, K)$ to $\mathcal{A}$;
- otherwise send $(sid, \mathcal{P}', \mathsf{pw}, \perp)$ to $\mathcal{A}$.

Figure 1: UC PAKE variants: The original PAKE functionality $\mathcal{F}_{\mathsf{pwKE}}$ of Canetti et al. [15] is the version with all gray text omitted. The lazy-extraction PAKE functionality $\mathcal{F}_{\mathsf{lePAKE}}$ [2] includes everything, and the variant of $\mathcal{F}_{\mathsf{lePAKE}}$ used in this work includes everything but the dashed box.

Definition 3.2 (Strong simultaneous CDH problem (sSDH))**.** Let $\mathcal{G}$ be a cyclic group and (X, g_1, g_2) sampled uniformly from $(\mathcal{G} \setminus \{I_{\mathcal{G}}\})^3$. Given access to oracles $\mathsf{DDH}(g_1, X, \cdot, \cdot)$ and $\mathsf{DDH}(g_2, X, \cdot, \cdot)$, provide $(Y, K_1, K_2) \in (\mathcal{G} \setminus \{I_{\mathcal{G}}\}) \times \mathcal{G} \times \mathcal{G}$ s. th. $\mathsf{DDH}(g_1, X, Y, K_1) = \mathsf{DDH}(g_2, X, Y, K_2) = 1$

As a cryptographic assumption sSDH above is justified since sSDH is implied by the gap simultaneous Diffie-Hellman assumption [39, 2], which allows for unlimited (i.e., with no fixed input) access to a DDH oracle. Lastly, we state a variant of the sSDH assumption where generators are sampled according to some probability distribution. Looking ahead, we require this variant since in CPace parties derive generators by applying a map which does not implement uniform sampling from the group. We state the non-uniform variant of sSDH for arbitrary probability distributions and investigate its relation to "uniform" sSDH afterwards.

With $\mathbf{Adv}^{\mathsf{sCDH}}_{\mathcal{B}_{\mathsf{sCDH}}}(\mathcal{G})$ and $\mathbf{Adv}^{\mathsf{sSDH}}_{\mathcal{B}_{\mathsf{sSDH}}}(\mathcal{G})$, we denote the probabilities that adversarial algorithms $\mathcal{B}_{\mathsf{sSDH}}$ and $\mathcal{B}_{\mathsf{sSDH}}$ having access to the restricted DDH oracles provide a solution for the sCDH and sSDH problems respectively in $\mathcal{G}$ when given a single randomly drawn challenge.

Definition 3.3 (Strong simultaneous non-uniform CDH problem ($\mathcal{D}_{\mathcal{G}}$-sSDH))**.** Let $\mathcal{G}$ be a group

and $\mathcal{D}_{\mathcal{G}}$ be a probability distribution on $\mathcal{G}$. The strong simultaneous non-uniform CDH problem $\mathcal{D}_{\mathcal{G}}$-sSDH is defined as the sSDH problem but with (X, g_1, g_2) sampled using $\mathcal{U}_{\mathcal{G}} \times \mathcal{D}_{\mathcal{G}} \times \mathcal{D}_{\mathcal{G}}$, where $\mathcal{U}_{\mathcal{G}}$ denotes the uniform distribution on $\mathcal{G}$.

Clearly, $\mathcal{U}_{\mathcal{G}\setminus\{I_{\mathcal{G}}\}}$-sSDH is equivalent to sSDH. We show that hardness of uniform and non-uniform sSDH are equivalent given that the distribution allows for probabilistic polynomial time (PPT) rejection sampling, which we now formalize.

Definition 3.4 (Rejection sampling algorithm for $(\mathcal{G}, \mathcal{D}_{\mathcal{G}})$)**.** Let $\mathcal{G}$ be a group and $\mathcal{D}_{\mathcal{G}}$ be a probability distribution on $\mathcal{G}$. With $\mathcal{D}_{\mathcal{G}}(g)$ we denote the probability for point g. Let RS be a probabilistic algorithm taking as input elements $g \in \mathcal{G}$ and outputting $\perp$ or a value $\neq\perp$. Then RS is called a *rejection sampling algorithm for* $(\mathcal{G}, \mathcal{D}_{\mathcal{G}})$ if there is a scaling factor k such that $Pr[\mathsf{RS}(g) \neq \perp] = k \cdot \mathcal{D}_{\mathcal{G}}(g)$ for $g \in \mathcal{G}$.

Informally RS is a probabilistic algorithm which accepts (output different from $\perp$) or rejects (output $\perp$) a candidate point. When queried multiple times on the same input $g \in \mathcal{G}$, the probability that g will be accepted or rejected models a scaled distribution that is proportional to $\mathcal{D}_{\mathcal{G}}$. In this paper, we are interested in rejection samplers with "good" acceptance rate, such that they can be efficiently used to sample elements from the scaled distribution. We formalize the acceptance rate as follows.

Definition 3.5 (Acceptance rate of a rejection sampler for $(\mathcal{G}, \mathcal{D}_{\mathcal{G}})$)**.** Let $\mathcal{G}$ be a group and $\mathcal{D}_{\mathcal{G}}$ be a probability distribution on $\mathcal{G}$. Let RS be a rejection sampling algorithm for $(\mathcal{G}, \mathcal{D}_{\mathcal{G}})$. Let $g_i \in \mathcal{G}$ be a sequence of m uniformly drawn points and $r_i = \mathsf{RS}(g_i)$. Then RS is said to have an acceptance rate of $(1/n)$ if the number of accepted points with $r_i \neq \perp$ converges to m/n when $m \to \infty$.

Using these definitions, we are able to prove that given some assumptions on the distribution $\mathcal{D}_{\mathcal{G}}$ hardness of sSDH and $\mathcal{D}_{\mathcal{G}}$-sSDH are equivalent up to the additional PPT computational effort for the rejection sampling algorithm.

Theorem 3.6 (sSDH $\iff$ $\mathcal{D}_{\mathcal{G}}$-sSDH)**.** *Let $\mathcal{G}$ be a cyclic group of order p and $\mathcal{D}_{\mathcal{G}}$ a probability distribution on $\mathcal{G}$. If there exists a PPT rejection sampler RS for $(\mathcal{G}, \mathcal{D}_{\mathcal{G}})$ with acceptance rate $(1/n)$ then the probability of PPT adversaries against $\mathcal{D}_{\mathcal{G}}$-sSDH and sSDH of solving the respectively other problem differs by at most $(2\mathcal{D}(I_{\mathcal{G}}) + (1/p))$ and solving sSDH with the help of a $\mathcal{D}_{\mathcal{G}}$-sSDH adversary requires at most $2n$ executions of RS on average.*

Proof. **sSDH hard $\Rightarrow$ $\mathcal{D}_{\mathcal{G}}$ − sSDH hard:** Given an adversary $\mathcal{B}_{\mathcal{D}_{\mathcal{G}}-\mathsf{sSDH}}$ against $\mathcal{D}_{\mathcal{G}}$ − sSDH with non-negligible success probability ν, we show how to construct an adversary $\mathcal{A}_{\mathsf{sSDH}}$. On receiving an sSDH-challenge (X, g_1, g_2), first note that X is uniformly sampled from $\mathcal{G} \setminus \{I_{\mathcal{G}}\}$. $\mathcal{A}_{\mathsf{sSDH}}$ uniformly samples $r, s \in \mathbb{Z}_p$ until $\mathsf{RS}(g_1^r) \neq \perp$ and $\mathsf{RS}(g_2^s) \neq \perp$, which requires $2n$ calls to RS on average. $\mathcal{A}_{\mathsf{sSDH}}$ runs $\mathcal{B}_{\mathcal{D}_{\mathcal{G}}-\mathsf{sSDH}}$ on input (X, g_1^r, g_2^s). If $\mathcal{B}$ queries $\mathsf{DDH}(g_1^r, X, Z, L)$, $\mathcal{A}$ queries his own oracle with $\mathsf{DDH}(g_1, X, Z, L^{1/r})$ and relays the answer to $\mathcal{B}$ (queries g_2^s are handled analogously). On receiving (Y, K_1, K_2) from $\mathcal{B}_{\mathcal{D}_{\mathcal{G}}-\mathsf{sSDH}}$, $\mathcal{A}_{\mathsf{sSDH}}$ provides $(Y, K_1^{1/r}, K_2^{1/s})$ as solution in his sSDH experiment.

As RS is a rejection sampler for $\mathcal{D}_{\mathcal{G}}$, (X, g_1^r, g_2^s) is a random $\mathcal{D}_{\mathcal{G}}$ − sSDH challenge, and thus $\mathcal{B}$ solves it with probability ν. If $\mathcal{B}$ provides a solution, then $\mathcal{A}_{\mathsf{sSDH}}$ succeeds in solving his own challenge unless g_1^r or $g_2^s = I_{\mathcal{G}}$ or $g_1^r = g_2^s$ which occurs at most with probability $(2\mathcal{D}_{\mathcal{G}}(I_{\mathcal{G}}) + 1/p)$. As RS executes in PPT, $\mathcal{A}_{\mathsf{sSDH}}$ is PPT, uses $(2n)$ calls to RS on average and succeeds with probability $\nu(1 - 2\mathcal{D}_{\mathcal{G}}(I_{\mathcal{G}}) - 1/p)$, which is non-negligible since ν is.

sSDH hard $\Rightarrow$ $\mathcal{D}_\mathcal{G}-$sSDH hard: Given an adversary $\mathcal{A}_{\mathsf{sSDH}}$ against sSDH with non-negligible probability μ we show how to construct a $\mathcal{D}_\mathcal{G}-$sSDH adversary $\mathcal{B}_{\mathcal{D}_\mathcal{G}-\mathsf{sSDH}}$. On receiving a $\mathcal{D}_\mathcal{G}-$sSDH challenge (X, g_1, g_2), $\mathcal{B}$ samples $r, s \in \mathbb{Z}_p \setminus 0$ and starts $\mathcal{A}_{\mathsf{sSDH}}$ on input (X, g_1^r, g_2^s). DDH oracle queries are handled the same as above. On receiving (Y, K_1, K_2) from $\mathcal{A}_{\mathsf{sSDH}}$, $\mathcal{B}$ provides $(Y, K_1^{1/r}, K_2^{1/s})$ as solution to his own challenge.

If $\mathcal{A}$ is successful, then $\mathcal{B}$ succeeds unless either g_1 or $g_2 = I_\mathcal{G}$ or $g_1^r = g_2^s$ which occurs at most with probability $(2\mathcal{D}_\mathcal{G}(I_\mathcal{G}) + 1/p)$. Thus, $\mathcal{B}$ is a PPT adversary against $\mathcal{D}_\mathcal{G} - \mathsf{sSDH}$ succeeding with non-negligible probability $\mu(1 - 2\mathcal{D}_\mathcal{G}(I_\mathcal{G}) - 1/p)$. □

Informally, the assumptions sSDH and $\mathcal{D}_\mathcal{G}-$sSDH become equivalent if stepping over an element that gets accepted in the sampling process becomes sufficiently likely for a randomly drawn sequence of candidates. Secondly, the probability of accidentally drawing the neutral element from $\mathcal{D}_\mathcal{G}$ needs to be negligible.

3.3 Transforming passwords to points on an elliptic curve

The generators of the Diffie-Hellman exchange in CPace are computed using a deterministic mapping function $\mathsf{Gen}(pw)$. For a given curve group $\mathcal{G}$ over a field $\mathbb{F}_q$, $\mathsf{Gen}(pw)$ is calculated with the help of either one ($\mathsf{Gen_{1MAP}}$) or two ($\mathsf{Gen_{2MAP}}$) invocations of a function $\mathsf{Map2Pt}_\mathcal{G} : \mathbb{F}_q \to \mathcal{G}$ and a hash function H_1 hashing to $\mathbb{F}_q$. For the sake of shortened notation, we will drop the $\mathcal{G}$ subscript where the group is obvious from the context. In both cases, security of CPace relies on Map2Pt meeting the requirements from this section. Informally, we first require Map2Pt to be "invertible". That is, for any point on the image of the map, there must be an efficient algorithm that outputs all preimages in $\mathbb{F}_q$ of $\mathsf{Map2Pt}_\mathcal{G}$ for a given group element g. We use the notation $\mathsf{Map2Pt}_\mathcal{G}.\mathsf{PreImages}(g)$. Details on how such an inversion algorithm can be efficiently implemented for various elliptic curve groups are given in [21, 11, 13, 30] and references therein. Secondly, a bound for the maximum number of preimages $n_{\max}$ that $\mathsf{Map2Pt}_\mathcal{G}$ maps to the same element must be known and this $n_{\max}$ bound needs to be small (we use the notation $\mathsf{Map2Pt}_\mathcal{G}.n_{\max}$ for the bound that applies for a given $\mathsf{Map2Pt}_\mathcal{G}$ function and group $\mathcal{G}$). This is needed in order to construct a rejection sampling algorithm whose acceptance rate must depend on $n_{\max}$.

Definition 3.7. Let $\mathcal{G}$ be a group of points on an elliptic curve over a field $\mathbb{F}_q$. Let $\mathsf{Map2Pt} : \mathbb{F}_q \to \mathcal{G}$ be a deterministic function. Then $\mathsf{Map2Pt}(\cdot)$ is called *probabilistically invertible with at most n_{max} preimages* if there exists a probabilistic polynomial-time algorithm $(r_1, \ldots, r_{n_g}) \leftarrow \mathsf{Map2Pt.PreImages}(g)$ that outputs all n_g values $r_i \in \mathbb{F}_q$ such that $g = \mathsf{Map2Pt}(r_i)$ for any $g \in \mathcal{G}$; and $\forall g \in \mathcal{G}$, $n_{\max} \geq n_g \geq 0$.

For a map Map2Pt that fulfills the previous definition with a bound for the numbers of preimages $\mathsf{Map2Pt}.n_{\max}$, we define an "inversion algorithm" $\mathsf{Map2Pt}^{-1} : \mathcal{G} \to \mathbb{F}_q$ that, on input $g \in \mathcal{G}$, returns one of potentially many preimages of g under Map2Pt if a biased coin comes up heads. If the coin comes up tails, the algorithm outputs failure. The "inversion algorithm" also serves as rejection sampling algorithm for the distribution $\mathcal{D}_\mathcal{G}$ that is produced by $\mathsf{Map2Pt}(r)$ for uniformly distributed inputs $r \in \mathbb{F}_q$:

Lemma 3.8. *Let $\mathsf{Map2Pt} : \mathcal{G} \to \mathbb{F}_q$ be probabilistically invertible with at most $\mathsf{Map2Pt}.n_{max}$ preimages and let $\mathcal{D}_\mathcal{G}$ denote the distribution it induces on $\mathcal{G}$. Then Algorithm 1 is a PPT rejection sampler for $(\mathcal{G}, \mathcal{D}_\mathcal{G})$ with average acceptance rate $(|\mathbb{F}_q|/|\mathcal{G}|)/\mathsf{Map2Pt}.n_{max}$.*

Algorithm 1 $\mathsf{Map2Pt}^{-1} : \mathcal{G} \longrightarrow \mathbb{F}_q \cup \{\perp\}$

On input $g \in \mathcal{G}$: Sample i uniformly from $\{1, \ldots, \mathsf{Map2Pt}.n_{\max}\}$; Then obtain $n_g \in \{0, \ldots, \mathsf{Map2Pt}.n_{\max}\}$ pre-images $(r_1, \ldots, r_m) \leftarrow \mathsf{Map2Pt.PreImages}(g)$; If $n_g < i$ return $\perp$, else return r_i.

Proof. We first define the average number of preimages $n_{\max} \geq \bar{n} \geq 1$ as the quotient of the order of the field $\mathbb{F}_q$ and the number of points on the image of the map, i.e., $\bar{n} = |\mathbb{F}_q|/|\text{support}(\mathcal{D}_{\mathcal{G}})|$. When drawing an element g uniformly from $\mathcal{G}$, the probability that the number of preimages n_g for g is nonzero is given by the quotient of the order of the support of $\mathcal{D}_{\mathcal{G}}$ and the order of the group. By the definition of $\bar{n}$ above this is $|\mathbb{F}_q|/(\bar{n}|\mathcal{G}|)$.

For any point on the map with a nonzero number n_g of preimages, Algorithm 1 returns a result $\neq\perp$ with probability $n_g/n_{\max}$. As the average value for the number of preimages for any point on the image of the map is $\bar{n}$, the average acceptance rate is $(|\mathbb{F}_q|/(\bar{n}|\mathcal{G}|)) \cdot \bar{n}/n_{\max} = (|\mathbb{F}_q|/|\mathcal{G}|)/n_{\max}$. □

Use of $\mathsf{Map2Pt}^{-1}$ for uniformly sampling field elements from $\mathbb{F}_q$. As Map2Pt is deterministic, each point g from $\mathcal{G}$ is characterized by the number of preimages n_g for Map2Pt in $\mathbb{F}_q$ with $n_{\max} \geq n_g \geq 0$. When generating points $\mathsf{Map2Pt}(s) \in \mathcal{G}$ for uniformly sampled field elements $s \leftarrow_{\mathrm{R}} \mathbb{F}_q$, the probability of obtaining a given point g is (n_g/q) and can only take the values of zero or integer multiples of $1/q$ up to $n_{\max}/q$. In order to compensate for this, $\mathsf{Map2Pt}^{-1}$ is constructed such that the probability of returning $r \neq\perp$ for a point g increases proportionally with n_g making any actually produced field element $r \neq\perp$ equally likely in $\mathbb{F}_q$. As a result, we can use $\mathsf{Map2Pt}^{-1}$ for transforming a sequence of uniformly sampled group elements $g_l \in \mathcal{G}$ to a sequence of uniformly sampled field elements $r_l \in \mathbb{F}_q$

Corollary 3.9. *Let Map2Pt be a probabilistically invertible map with at most $\mathsf{Map2Pt}.n_{max}$ preimages and let $g_l \leftarrow_R \mathcal{G}$. Then $r_l \leftarrow \mathsf{Map2Pt}^{-1}(g_l)$ outputs results $r_l \neq\perp$ with probability $p \geq (|\mathbb{F}_q|/|\mathcal{G}|)/\mathsf{Map2Pt}.n_{max}$ and the distribution of outputs $r_l \neq\perp$ is uniform in $\mathbb{F}_q$.*

Moreover as the collision probability when drawing two elements r_a, r_b from $\mathbb{F}_q$ is $1/q$ and as there are at most $n_{\max}$ values s_l generating the same group element $g = \mathsf{Map2Pt}(s_l)$ the collision probability for $g_a = \mathsf{Map2Pt}(r_a)$ and $g_b = \mathsf{Map2Pt}(r_b)$ is increased at most by $n_{\max}^2$.

Corollary 3.10. *When sampling two field elements $r_a, r_b \leftarrow_R \mathbb{F}_q$ uniformly, we have $\mathsf{Map2Pt}(r_a) = \mathsf{Map2Pt}(r_b)$ with a probability of at most n_{max}^2/q.*

4 The CPace protocol

The CPace protocol [25] is a SPEKE-type protocol [34] allowing parties to compute a common key via a Diffie-Hellman key exchange with password-dependent generators. The blueprint of the protocol is depicted in Fig. 2. Informally, a party $\mathcal{P}$ willing to establish a key with party $\mathcal{P}'$ first computes a generator g from a password pw. Next, $\mathcal{P}$ generates an element $Y_a = g^{y_a}$ from a secret value y_a sampled at random and sends it to $\mathcal{P}'$. Upon receiving a value Y_b from $\mathcal{P}'$, $\mathcal{P}$ then computes a Diffie-Hellman key $K = (Y_b)^{y_a} = g^{y_a y_b}$ and aborts if K equals the identity element. Finally, it computes the session key as the hash of K and the exchanged values Y_a and Y_b.

In order to allow for efficient instantiations over different types of groups, most of which are elliptic curves, we present the CPace protocol in form of a blueprint CPace[Gen, ScMul, ScMulVf, ScSam]

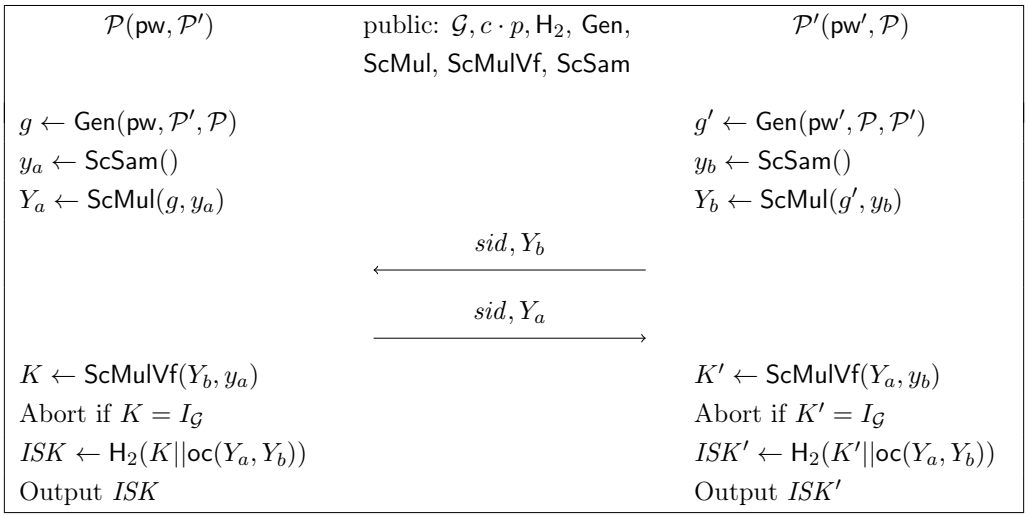

Figure 2: **Above:** Blueprint protocol CPace[Gen, ScMul, ScMulVf, ScSam] requiring group $\mathcal{G}$ of order $c \cdot p$ with prime p and algorithms for DH generator computation (Gen), exponentiation (ScMul, ScMulVf) and scalar sampling (ScSam). $\mathsf{H}_2 : \{0,1\}^* \to \{0,1\}^\lambda$ denotes a hash function. This blueprint can capture most artifacts of current CPace implementations.
Below: "Basic" CPace $\mathrm{CPace}_{\mathsf{base}}$ with $c = 1$, generators computed from hash function $\mathsf{H}_\mathcal{G} : \{0,1\}^* \to \mathcal{G}$ and canonical exponentation, point verification and sampling.

$\mathrm{CPace}_{\mathsf{base}} := \mathrm{CPace}[\mathsf{Gen}_{\mathsf{RO}}, \mathsf{ScMul}_{\mathsf{base}}, \mathsf{ScMulVf}_{\mathsf{base}}, \mathsf{ScSam}_p]$

$\mathsf{Gen}_{\mathsf{RO}}(\mathsf{pw}, \mathcal{P}, \mathcal{P}')$:	$\mathsf{ScMul}_{\mathsf{base}}(g, y)$:	$\mathsf{ScMulVf}_{\mathsf{base}}(g, y)$:	$\mathsf{ScSam}_p()$:
return $\mathsf{H}_\mathcal{G}(\mathsf{pw}\|\|\mathsf{oc}(\mathcal{P}, \mathcal{P}'))$	return g^y	if $g \notin \mathcal{G}$: return $I_\mathcal{G}$ else: return g^y	$y \leftarrow_\mathrm{R} \{1, \ldots, p\}$ return y

in Fig. 2 that provides the following generalizations: (1) The blueprint uses a generic algorithm $\mathsf{Gen} : \mathcal{D} \times PID \times PID \to \mathcal{G}$ that turns a password from dictionary $\mathcal{D}$ and two party identifiers into a group element. For correctness we require that the order in which party identifiers are input to Gen does not influence the output, i.e., $\mathsf{Gen}(\mathsf{pw}, \mathcal{P}, \mathcal{P}') = \mathsf{Gen}(\mathsf{pw}, \mathcal{P}', \mathcal{P})$ for all $\mathcal{P}, \mathcal{P}' \in PID$ and $\mathsf{pw} \in \mathcal{D}$; (2) The computation of the y_i and Y_i values is done with generic algorithms for sampling ($\mathsf{SamSc} : 0, 1^* \to 0, 1^*$) and scalar multiplication ($\mathsf{ScMul} : \mathcal{G} \times \mathbb{Z}_{|\mathcal{G}|} \to \mathcal{G}$); (3) The Diffie-Hellman key is computed with another generic algorithm $\mathsf{ScMulVf} : \mathcal{G} \times \mathbb{Z}_{|\mathcal{G}|} \to \mathcal{G}$, in order to allow for additional point verification that is necessary on some curves (but not on all) to protect against trivial attacks; (4) the blueprint protocol uses an ordered concatenation function oc so that messages can be sent in any order and parties do not have to play a specific initiator or responder role(see Appendix D). In the remainder of the paper, we will instantiate the CPace blueprint in various ways, by specifying a set of concrete algorithms Gen, ScMul, ScMulVf, ScSam.

On the necessity of point verification. Many elliptic curve scalar multiplication algorithms will work correctly independent whether the input operand encodes a point on the correct curve or not. As a consequence if group membership is not correctly verified by an implementation various attack scenarios become feasible. An active attacker may for instance provide a point on a curve of low order on which the discrete-logarithm problem could be solved. The threat for real-world

implementations is that this serious error might remain undetected as the corresponding verification event is never generated in communications of honest parties. In order to make CPace resilient to this type of attack and implementation pitfalls, [25] suggested to first restrict invalid curve attacks to the quadratic twist (by using a single-coordinate Montgomery ladder) and then choose a curve where also the twist has a large prime-order subgroup and invalid curve attacks become impossible. The CPace draft [26] highlights this aspect on the protocol specification level by introducing a $\mathsf{ScMulVf}$ function which is specified to include point verification.

5 Security of Simplified CPace

In this Section, as a warm-up, we analyze security of a "basic" variant of CPace, which we call $\text{CPace}_{\mathsf{base}}$ and which is depicted in Fig. 2. We instantiate Gen with a hash function $\mathsf{H}_{\mathcal{G}}$ that hashes onto the group $\mathcal{G}$. This way, parties compute generators as $g \leftarrow \mathsf{H}_{\mathcal{G}}(\mathsf{pw})$. Further, we assume $\mathcal{G}$ to be a multiplicatively written group of prime order p where group membership is efficiently decidable. We instantiate $\mathsf{ScMul}(g, y) := g^y$ as exponentiation, and $\mathsf{ScMulVf}(g, y)$ such that it returns the neutral element if g is not in the group and g^y otherwise, and SamSc with uniform sampling from $\{1 \dots p\}$. A formal description of the protocol is given in Fig. 2, where the blueprint protocol is instantiated with the algorithms at the bottom of the Figure.

For clarity, we give a UC execution of $\text{CPace}_{\mathsf{base}}$ in Fig. 15 and prove its security properties in the following theorem.

Theorem 5.1 (Security of $\text{CPace}_{\mathsf{base}}$)**.** *Let $\lambda, p \in \mathbb{N}$ with p prime. Let $\mathcal{G}$ be a group of order p, and let $\mathsf{H}_1 : \{0,1\}^* \to \mathcal{G}, \mathsf{H}_2 : \{0,1\} \to \{0,1\}^\lambda$ be two hash functions. If the sCDH and sSDH problems are hard in $\mathcal{G}$, then protocol $\text{CPace}_{\mathsf{base}}$ depicted in Fig. 2 UC-emulates $\mathcal{F}_{\mathsf{lePAKE}}$ in the random-oracle model with respect to adaptive corruptions when both hash functions are modeled as random oracles. More precisely, for every adversary $\mathcal{A}$, there exist adversaries $\mathcal{B}_{\mathsf{sSDH}}$ and $\mathcal{B}_{\mathsf{sCDH}}$ against the strong CDH (sCDH) and strong simultaneous CDH (sSDH) problems such that*

$$\begin{aligned} &|Pr[Real_{\mathcal{Z}}(\text{CPace}_{\mathsf{base}}, \mathcal{A})] - Pr[Ideal_{\mathcal{Z}}(\mathcal{F}_{\mathsf{lePAKE}}, \mathcal{S})| \\ &\qquad \leq l_{\mathsf{H}_1}^2/p + 2l_{\mathsf{H}_1}^2 \mathbf{Adv}^{\mathsf{sSDH}}_{\mathcal{B}_{\mathsf{sSDH}}}(\mathcal{G}) + \mathbf{Adv}^{\mathsf{sCDH}}_{\mathcal{B}_{\mathsf{sCDH}}}(\mathcal{G}) \end{aligned}$$

where l_{H_1} denotes the number of H_1 queries made by the adversary $\mathcal{A}$ and the simulator $\mathcal{S}$ is depicted in Fig. 3.

Sketch. The main idea of the simulation is to fix a secret generator $g \in \mathcal{G}$ and carry out the simulation with respect to g. Messages of honest parties are simulated as g^z for a fresh exponent z. Queries $\mathsf{H}_1(\mathsf{pw})$ are answered with g^r for a freshly chosen "trapdoor" r. The simulator might learn an honest party's password via adaptive corruption or via an adversarial password guess. The simulator can now adjust the simulation in retrospective to let the honest party use $g^r = \mathsf{H}_1(\mathsf{pw})$ by claiming the party's secret exponent to be zr^{-1}. This already concludes simulation of honest parties without passwords. Adversarial password guesses can be read from $\mathcal{A}$ injecting X (or, similarly, Y) and then querying $\mathsf{H}_2(K||X||Y)$ with K being a correctly computed key w.r.t some generator g^r provided by the simulation. $\mathcal{S}$ can now read the guessed password from the H_1 list, and submit it as password guess to $\mathcal{F}_{\mathsf{lePAKE}}$. In case of success, the simulator sets the key of the honest party to $\mathsf{H}_2(K||X||Y)$.

The simulation is complicated by the order of honest parties' outputs (which are generated upon receipt of the single message) and the adversary's computation of final session keys via H_2 queries.

The simulator $\mathcal{S}$ ($\mathbf{G}_2$)samples and stores a generator $g \leftarrow \mathcal{G}$.

On $(\mathsf{NewSession}, sid, P_i, P_j)$ from $\mathcal{F}_{\mathsf{lePAKE}}$:
($\mathbf{G}_4$)sample $z_i \leftarrow_{\mathrm{R}} \mathbb{Z}_p$
($\mathbf{G}_4$)set $Y_i \leftarrow g^{z_i}$; store $(P_i, P_j, z_i, Y_i, \perp)$
($\mathbf{G}_4$)send Y_i to $\mathcal{A}$ intended to P_j

On Z^* from $\mathcal{A}$ as msg to (sid, P_i):
($\mathbf{G}_4$)if Z^* is adversarially generated
($\mathbf{G}_4$)and $Z^* \in \mathcal{G} \setminus I_{\mathcal{G}}$:
send $(\mathsf{RegisterTest}, sid, P_i)$ to $\mathcal{F}_{\mathsf{lePAKE}}$

Upon P_i receiving (sid, Y_j) with $Y_j \in \mathcal{G}$ from P_j:
($\mathbf{G}_4$)retrieve record $(P_i, P_j, z_i, Y_i, *)$
($\mathbf{G}_4$)if $\exists$ records ($\mathbf{G}_5$)$(\mathsf{H}_1, \mathsf{pw}||\mathsf{oc}(P_i, P_j), r, r^{-1}, G)$, $(\mathsf{H}_2, K||\mathsf{oc}(Y_i, Y_j), ISK)$ s.th. $K = Y_j^{z_i r^{-1}}$:
($\mathbf{G}_5$) store (guess, G, Y_j); abort if $\exists$ record $(\mathsf{guess}, G', Y_j)$ with $G \neq G'$;
($\mathbf{G}_4$) send $(\mathsf{TestPwd}, sid, P_i, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{lePAKE}}$;
($\mathbf{G}_4$) send $(\mathsf{NewKey}, sid, P_i, ISK)$ to $\mathcal{F}_{\mathsf{lePAKE}}$ and store $(P_i, P_j, z_i, Y_i, ISK)$
($\mathbf{G}_4$)else: sample a fresh random ISK' and send $(\mathsf{NewKey}, sid, P_i, ISK')$ to $\mathcal{F}_{\mathsf{lePAKE}}$

On $\mathsf{H}_1(\mathsf{pw}||\mathcal{P}||\mathcal{P}')$ from $\mathcal{A}$:
($\mathbf{G}_2$)if this is the first such query:
($\mathbf{G}_2$) sample $r \leftarrow_{\mathrm{R}} \mathbb{F}_p \setminus \{0\}$
($\mathbf{G}_3$) abort if $\exists$ rec. $(\mathsf{H}_1, *, r, *, *)$
($\mathbf{G}_2$) store $(\mathsf{H}_1, \mathsf{pw}||\mathcal{P}||\mathcal{P}', r, r^{-1}, g^r)$
($\mathbf{G}_2$)retrieve rec. $(\mathsf{H}_1, \mathsf{pw}||\mathcal{P}||\mathcal{P}', *, *, h)$
($\mathbf{G}_2$)reply with h

On msg $(\mathsf{AdaptiveCorruption}, sid, P_i)$ from $\mathcal{A}$ (instructed by $\mathcal{Z}$):
send $\mathsf{AdaptiveCorruption}, sid, P_i)$ to $\mathcal{F}_{\mathsf{lePAKE}}$
Upon receiving $(sid, P_j, \mathsf{pw}, ISK)$:
($\mathbf{G}_4$)if a msg $Y_i := g^{z_i}$ already sent to P_j:
($\mathbf{G}_4$) if $\exists$ rec. $(\mathsf{H}_1, \mathsf{pw}||\mathsf{oc}(P_i, P_j), r, r^{-1}, *)$: $y_i \leftarrow z_i r^{-1}$
($\mathbf{G}_4$) else: $r \leftarrow_{\mathrm{R}} \mathbb{Z}_p$; $y_i \leftarrow z_i r^{-1}$ and
($\mathbf{G}_4$) store $(\mathsf{H}_1, \mathsf{pw}||\mathsf{oc}(P_i, P_j), r, r^{-1}, g^r)$
($\mathbf{G}_4$) if $ISK \neq \perp$, let Y_j denote the message sent to P_i
($\mathbf{G}_4$) and store $(H_2, Y_j^{y_i}||\mathsf{oc}(Y_i, Y_j), ISK)$
($\mathbf{G}_4$) send $(sid, P_i, P_j, \mathsf{pw}, y_i, Y_i, Y_j^{y_i}, ISK)$ to $\mathcal{A}$

On $\mathsf{H}_2(K||Y_i||Y_j)$ from $\mathcal{A}$:
($\mathbf{G}_4$)if this is the first such query then
($\mathbf{G}_7$) if $\exists$ rec.$(P_i, P_j, z_i, Y_i, *), (P_j, P_i, z_j, Y_j, *), (\mathsf{H}_1, *, r, *)$ such that $K^r = g^{z_i z_j}$: abort;
($\mathbf{G}_4$) if $\nexists$ rec.$(P_i, P_j, *, Y_i, *)$ or $(P_j, P_i, *, Y_j, *)$, or if $Y_a||Y_b \neq \mathsf{oc}(Y_a, Y_b)$: $A \leftarrow_{\mathrm{R}} \{0,1\}^{2k}$;
($\mathbf{G}_4$) if $\exists$ records $(P_i, P_j, z_i, Y_i, ISK)$ with $ISK \neq \perp$ and ($\mathbf{G}_5$)$(\mathsf{H}_1, \mathsf{pw}||\mathsf{oc}(P_i, P_j), r, r^{-1}, G)$ s.th. $K = Y_j^{z_i r^{-1}}$:
($\mathbf{G}_5$) Record (guess, G, Y_j); abort if $\exists$ rec. $(\mathsf{guess}, G', Y_j)$ with $G \neq G'$.
($\mathbf{G}_5$) Send $(\mathsf{LateTestPwd}, sid, P_i, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{lePAKE}}$. Upon answer $\hat{K}$ set $A \leftarrow \hat{K}$;
($\mathbf{G}_4$) if $\exists$ $(P_j, P_i, z_j, Y_j, ISK)$ with $ISK \neq \perp$ and ($\mathbf{G}_5$)$(\mathsf{H}_1, \mathsf{pw}||\mathsf{oc}(P_j, P_i), r, r^{-1}, G)$ s.th. $K = Y_i^{z_j r^{-1}}$:
($\mathbf{G}_5$) Store (guess, G, Y_i); Abort if $\exists$ record $(\mathsf{guess}, G', Y_i)$ with $G \neq G'$;
($\mathbf{G}_5$) Send $(\mathsf{LateTestPwd}, sid, P_j, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{lePAKE}}$. Upon answer $\hat{K}$ set $A \leftarrow \hat{K}$
($\mathbf{G}_4$) if no matching H_1 records are found set $A \leftarrow_{\mathrm{R}} \{0,1\}^{2k}$
($\mathbf{G}_4$) finally, store $(\mathsf{H}_2, K||Y_i||Y_j, A)$ and reply with A
($\mathbf{G}_4$)else retrieve record $(\mathsf{H}_2, K||Y_i||Y_j, A)$ and reply with A

Figure 3: Simulator for $\mathsf{CPace}_{\mathsf{base}}$, with game numbers to indicate which game introduces a particular line of code (cf. full proof of Theorem 5.1 in Appendix A). For brevity we omit the session identifier *sid* from all records stored by the simulator.

If the key is generated by $\mathcal{F}_{\mathsf{lePAKE}}$ before $\mathcal{A}$ computes it via H_2 (which constitutes a password guess as detailed above), then $\mathcal{S}$ needs to invoke the $\mathsf{LateTestPwd}$ query of $\mathcal{F}_{\mathsf{lePAKE}}$ instead of $\mathsf{TestPwd}$. In case of a correct guess, this lets $\mathcal{S}$ learn the output key of the honest party, which $\mathcal{S}$ can then program into the corresponding H_2 query.

Finally, the simulation can fail in some cases. Firstly, $\mathcal{S}$ might find more than one password guess against an honest party with simulated message X. In this case, the simulation cannot continue since $\mathcal{F}_{\mathsf{lePAKE}}$ allows for only one password guess per party. In this case, however, $\mathcal{A}$ would provide (g^r, X, Y, K),$(g^{r'}, X, Y, K')$ which are two CDH tuples for passwords $\mathsf{pw}, \mathsf{pw}'$ with $g^r \leftarrow \mathsf{H}_1(\mathsf{pw}), g^{r'} \leftarrow \mathsf{H}_1(\mathsf{pw}')$. Provided that the simultaneous strong CDH assumption (sSDH, cf.

Definition 3.2) holds, this cannot happen. Here, the "strong" property, providing a type of DDH oracle, is required to help $\mathcal{S}$ identify CDH tuples among all queries to H_2. A second case of simulation failure occurs when $\mathcal{A}$ wants to compute a key of an uncorrupted session via a H_2 query. Since $\mathcal{S}$ does not know such keys, it would have to abort. Using a similar strategy as above, pseudorandomness of keys can be shown to hold under the strong CDH assumption, and thus the probability of $\mathcal{A}$ issuing such a H_2 query is negligible. The full proof can be found in Appendix A. □

Our Theorem 5.1 demonstrates that adaptive security of CPace can be proven with only minimal information included in the hashes, i.e., the first hash requires only the password and the final key derivation hash requires the Diffie-Hellman key and the unique protocol transcript. We detail now under which circumstances additional data such as session identifiers needs to be included in the hashes. We further note that adding additional inputs to hashes, such as the name of a ciphersuite that an application wants parties to agree on, does not harm security.

On multi-session security and hash domain separation Theorem 5.1 demonstrates that $\mathsf{CPace}_{\mathsf{base}}$ allows to securely turn a joint password into one key. In practice, one would of course want to exchange more than one key, and many parties will end up using the same password. If session identifiers are globally unique, then the UC composition theorem (more detailed, the composition theorem of UC with Joint State [16]) allows to turn Theorem 5.1 into a proof of "multi-session CPace" by simply appending the unique session identifiers to all hash function inputs. This ensures that hash domains of individual sessions are separated and the programming activities of the individual simulators do not clash. To summarize, we obtain a secure multi-session version of CPace by ensuring uniqueness of session identifiers and including them in hashes. In practice, this can be ensured by, e.g., agreeing on a joint session identifier to which both users contributed randomness and in which party identifiers are incorporated (see, e.g., [6]). The agreement needs to happen before starting CPace, but does not require secrecy and can thus potentially be piggy-backed to messages sent by the application. As a last note, applications might choose to add more values to hashes, for example to authenticate addresses or to ensure agreement on a ciphersuite. We stress that such additional values do not void our security analysis, but care still needs to be taken in order to protect against side-channel attacks.

5.1 Embedding CDH experiment libraries into the simulator

In this section, we discuss an alternative approach to carrying out reductions to cryptographic assumptions in the case of CPace/CDH. Both assumptions required by $\mathsf{CPace}_{\mathsf{base}}$, sCDH and sSDH, allow for an *efficient implementation* of experiments in the following sense: the secret exponents that are sampled by the experiment code (often also called the *challenger*) are sufficient for answering the restricted DDH queries allowed by both assumptions. An example for an assumption that does not allow for such efficient instantiation is, e.g., gap-CDH. In gap-CDH, the adversary is provided with a "full" DDH oracle that he can query on arbitrary elements, of which the experiment might not know an exponent for.

Due to this property, we can integrate implementations of the sCDH and sSDH experiments in the simulator's code. The simulator implements the DDH oracles on its own, and abort if at any time an oracle query solves the underlying assumption. We chose to integrate experiments as libraries (written as objects in python-style notation in Figure 4) into the simulator's code. This

```
# using python-style notation with self pointer s and _init_ constructor
def class sCDH:
  def _init_(s, g): s.g ← g; s.i ← 0; s.state← fresh;
  def sampleY(s):
    if s.i < 2: s.i+ = 1; sample s.y_i ←R F_p \ 0; return (s.g)^{s.y_i};
  def corrupt(s,X):
    for 1 ≤ m ≤ s.i:
      if (X = (s.g)^{s.y_m}): s.state← corrupt; return s.y_m;
  def DDH(s, g, Y, X, K):
    if (g ≠ s.g): return;
    if ({Y, X}={s.Y_1, s.Y_2}) and (s.state= fresh) and (K = (s.g)^{s.y_1·s.y_2}):
      abort("sCDH(g, Y_1, Y_2) solved")
    for 1 ≤ m ≤ s.i:
      if (Y = (s.g)^{s.y_m}): return (K = X^{s.y_m});
  def isValid(X): return (X ∈ G \ {I_G})

def class sSDH: # using python-style notation [ ] for list containers
  def _init_(s,sCdhExp): # Gets sCDH class; samples g; creates a sCDH instance
    s.g ←R G; s.scdh = sCdhExp(s.g); s.records =[ ]; s.guess = "yet no guess";
  def sampleY(s): return (s.scdh).sampleY();
  def isValid(s, X): return (s.scdh).isValid(X);
  def sampleH1(s):
    sample r ←R F_p \ {0};
    if (r, ∗) in s.records: abort("Hash to group collision");
    else: s.records.append((r, (s.g)^r)); return (s.g)^r;
  def corrupt(s, R, Y):
    if there is (r, R) in s.records: return (s.scdh).corrupt(Y^{1/r});
  def DDH(s,R, Y, X, K):
    if there is (r, R) in s.records:
      match ← (s.scdh).DDH(s.g, Y, X, K^{1/r});
      if match and (s.guess = "yet no guess"): (s.guess.g,s.guess.X)← (R, X);
      elif match and (s.guess.X = X) and (s.guess.g ≠ R):
        abort("sSDH problem (Y, R, s.guess.g) solved");
      return match;
```

Figure 4: Libraries implementing sCDH and sSDH experiments.

eases not only presentation but is also useful for analyzing variants of CPace that require slightly different assumptions.

The corresponding result for $\text{CPace}_{\mathsf{base}}$ is shown in Fig. 5 when the challenge-generating experiment $\mathsf{exp} \leftarrow \mathsf{sSDH}(\mathsf{sCDH})$ is used (Fig. 4). The instance of the sSDH object first samples a random generator as member $s.g$ and creates a member instance $s.scdh \leftarrow \mathsf{sCDH}(g)$ of the experiment for the sCDH problem. The sCDH member object produces a challenge consisting of two uniformly drawn group elements $Y_1 \leftarrow g^{y_1}, Y_2 \leftarrow g^{y_2}$. The limited DDH oracle provided by the sCDH assumption can only receive inputs w.r.t one of these elements, and thus it can be implemented efficiently

The simulator $\mathcal{S}(\mathsf{exp})$ is parametrized by an experiment class exp.

On $(\mathsf{NewSession}, sid, P_i, P_j)$ from $\mathcal{F}_{\mathsf{lePAKE}}$:
($\mathbf{G}_4$)set $Y_i \leftarrow \mathsf{exp.sampleY}()$;
($\mathbf{G}_4$)store $(P_i, P_j, Y_i, \perp)$;
($\mathbf{G}_4$)send Y_i to $\mathcal{A}$ intended to P_j;

On Z^* from $\mathcal{A}$ as msg to (sid, P_i):
($\mathbf{G}_4$)if Z^* is adversarially generated
($\mathbf{G}_4$)and $\mathsf{exp.isValid}(Z^*)$:
($\mathbf{G}_4$) send $(\mathsf{RegisterTest}, sid, P_i)$ to $\mathcal{F}_{\mathsf{lePAKE}}$

Upon P_i receiving (sid, Y_j) from P_j: retrieve record $(P_i, *, z_i, Y_i, *)$
($\mathbf{G}_1$)if not $\mathsf{exp.isValid}(Y_j)$: return;
($\mathbf{G}_4$)if $\exists$ records $(\mathsf{H}_1, \mathsf{pw}, h)$, $(\mathsf{H}_2, K||(\mathsf{oc}(Y_i, Y_j), ISK)$
($\mathbf{G}_4$)such that $\mathsf{exp.DDH}(h, Y_i, Y_j, K) = 1$:
($\mathbf{G}_4$) send $(\mathsf{TestPwd}, sid, P_i, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{lePAKE}}$
($\mathbf{G}_4$) send $(\mathsf{NewKey}, sid, P_i, ISK)$ to $\mathcal{F}_{\mathsf{lePAKE}}$ and store (P_i, P_j, Y_i, ISK)
($\mathbf{G}_4$)else sample a fresh random ISK' and send $(\mathsf{NewKey}, sid, P_i, ISK')$ to $\mathcal{F}_{\mathsf{lePAKE}}$
($\mathbf{G}_4$) # $\mathcal{F}_{\mathsf{lePAKE}}$ will discard ISK'

On $\mathsf{H}_1(\mathsf{pw}||P_i||P_j))$ from $\mathcal{A}$:
($\mathbf{G}_2$)if not $\exists$ record
($\mathbf{G}_2$) $(\mathsf{H}_1, \mathsf{pw}||P_i||P_j, h)$:
($\mathbf{G}_2$) $h \leftarrow \mathsf{exp.sampleH1}()$
($\mathbf{G}_2$) store $(\mathsf{H}_1, \mathsf{pw}||P_i||P_j, h)$
($\mathbf{G}_2$)lookup$(\mathsf{H}_1, \mathsf{pw}||P_i||P_j, h)$
($\mathbf{G}_2$)reply with h

On $(\mathsf{AdaptiveCorruption}, sid)$ from $\mathcal{A}$ as msg to P_i:
Lookup $(P_i, P_j, Y_i, *)$; send $(\mathsf{AdaptiveCorruption}, sid, P_i)$ to $\mathcal{F}_{\mathsf{lePAKE}}$ and obtain back (sid, pw, ISK);
($\mathbf{G}_4$)if a message Y_i was already sent to P_j, then:
($\mathbf{G}_4$) query $\mathsf{H}_1(\mathsf{pw})$ and retrieve record $(\mathsf{H}_1, \mathsf{pw}, h)$
($\mathbf{G}_4$) $y_i \leftarrow \mathsf{exp.corrupt}(h, Y_i)$;
($\mathbf{G}_4$) if $ISK \neq \perp$:
($\mathbf{G}_4$) let Y_j denote the message sent to P_i
($\mathbf{G}_4$) store $(\mathsf{H}_2, sid||Y_j^{y_1}||\mathsf{oc}(Y_i, Y_j), ISK)$
($\mathbf{G}_4$) send $(sid, P_i, \mathsf{pw}, y_i, Y_i, Y_j^{y_i}, ISK)$ to $\mathcal{A}$

On $\mathsf{H}_2(sid||K||Y_i||Y_j)$ from $\mathcal{A}$:
($\mathbf{G}_4$)lookup $(\mathsf{H}_2, sid||K||Y_i||Y_j, h)$ and send h if it exists;
($\mathbf{G}_4$)else if this is the first such query:
($\mathbf{G}_4$) if there are no records $(P_i, P_j, Y_i, *)$ or $(P_j, P_i, Y_j, *)$, or if $Y_a||Y_b \neq \mathsf{oc}(Y_a, Y_b)$:
($\mathbf{G}_4$) sample $A \leftarrow \{0,1\}^{2k}$;
($\mathbf{G}_4$) if $\exists$ records (P_i, P_j, Y_i, ISK) with $ISK \neq \perp$ and $(\mathsf{H}_1, \mathsf{pw}, h)$
($\mathbf{G}_4$) such that $\mathsf{exp.DDH}(h, Y_i, Y_j, K) = 1$:
($\mathbf{G}_5$) send $(\mathsf{LateTestPwd}, sid, P_i, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{lePAKE}}$. Upon answer $\hat{K}$ set $A \leftarrow \hat{K}$
($\mathbf{G}_4$) if $\exists$ records (P_j, P_i, Y_j, ISK) with $ISK \neq \perp$ and $(\mathsf{H}_1, \mathsf{pw}, h)$
($\mathbf{G}_4$) such that $\mathsf{exp.DDH}(h, Y_j, Y_i, K) = 1$:
($\mathbf{G}_5$) send $(\mathsf{LateTestPwd}, sid, P_j, \mathsf{pw})$ to $\mathcal{F}_{\mathsf{lePAKE}}$. Upon answer $\hat{K}$ set $A \leftarrow \hat{K}$
($\mathbf{G}_4$) if no matching H_1 records are found set $A \leftarrow \{0,1\}^{2k}$
($\mathbf{G}_4$) finally, store $(\mathsf{H}_2, sid||K||Y_i||Y_j, A)$ and reply with A

Figure 5: Generic simulator for different CPace variants, embedding challenges generated by the experiment object exp. The simulator for $\mathrm{CPace}_{\mathsf{base}}$ is obtained when using $\mathsf{exp} \leftarrow \mathsf{sSDH}(\mathsf{sCDH})$ from Fig. 4.

using secret exponents y_1, y_2. If a correct CDH solution $g, Y_1, Y_2, g^{y_1 \cdot y_2}$ is provided, the sCDH object aborts. In its implementation for H_1, the sSDH object samples random new generators as $R \leftarrow (s.g)^r$ which will be used for simulating password-dependent base points and uses the $s.scdh$ member and the known exponent r for answering DDH queries by use of the $s.scdh.\mathsf{DDH}$ function. The corrupt queries are implemented likewise and forwarded to the $s.scdh$ member object. The simulator from Fig. 3 is adapted to call the experiment. As an example, honest parties' messages

are simulated by calling the challenge sampling procedure exp.sampleY() from sSDH which itself calls the corresponding function from its sCDH member.

Proving indistinguishability. With this simulation approach, a proof consists in demonstrating that ideal and real world executions are indistinguishable except for events in which the experiment libraries abort because a challenge was correctly answered. Compared to our proof of Theorem 5.1, the indistinguishability argument becomes simpler because the reduction strategies to both CDH-type assumptions are already embedded in the corresponding assumption experiment libraries. Losses such as the factor $2l_{\mathsf{H}_1}^2$ in the reduction to sSDH in game $\mathbf{G}_5$ translate to libraries producing more than one challenge per simulation run, as is the case for the sSDH experiment from Fig. 5. Altogether, the simulation with integrated CDH experiment libraries is an alternative approach of proving Theorem 5.1, as we formalize in the following.

Theorem 5.2 (Alternative simulation for Theorem 5.1)**.** *The simulator depicted in Fig. 5 is a witness for the UC emulation statement in Theorem 5.1*

Proof sketch. The output distribution of the simulator $\mathcal{S}$ from Fig. 5 is indistinguishable from the one of the simulator from Fig. 3 as it is obtained from internal restructuring. $\mathcal{S}$ aborts if either the sSDH or the sCDH experiment class aborts, which occurs iff a correct solution has been provided to the experiment implementation or a H_1 collision is observed. These cases coincide with the abort cases in the proof of Theorem 5.1. As the sSDH object outputs $2l_{\mathsf{H}_1}^2$ different challenges and as it is sufficient for $\mathcal{Z}$ to provide a solution to one of these challenges for distinguishing both worlds, the advantage for solving the sSDH problem needs to be multiplied by this factor, thus reproducing the bounds from Theorem 5.1. □

Advantages of embedding libraries in the simulation. To clarify, the approach presented in this section does *not* allow to prove stronger security statements. As demonstrated above, it is merely an alternative way of presenting security proofs in the UC framework or other simulation-based frameworks, and it works whenever the underlying cryptographic assumptions are efficiently implementable. However, we believe that the approach has its merits especially in the following dimensions.

- **Modular security analysis.** Slight modifications in the protocol might require to change the cryptographic assumption. As long as the public interface does not change, our approach allows to switch between assumptions by simply calling a different library. Cryptographers then need to only analyze this "local" change in the simulation, which prevents them from re-doing the whole indistinguishability argument.
- **Presentation of reduction strategies.** In normal game-based indistinguishability arguments [41], reductions to cryptographic assumptions are hidden within side-long proofs. With our approach, the reduction strategy is depicted in clear code as part of the simulator's code. This makes checking of proofs easier not only for readers but also might make simulation-based proofs more accessible to automated verification.

In this paper, our motivation is the first dimension. In the upcoming section, the library-based approach will turn out to be extremely useful to analyze the various variants of CPace that stem from (efficiency-wise) optimized implementations on different elliptic curves.

$\mathrm{CPace}_{\mathsf{1MAP}} := \mathrm{CPace}[\mathsf{Gen}_{\mathsf{1MAP}}, \mathsf{ScMul}_{\mathsf{base}}, \mathsf{ScMulVf}_{\mathsf{base}}, \mathsf{ScSam}_p]$			
$\mathsf{Gen}_{\mathsf{1MAP}}(\mathsf{pw}, P_i, P_j)$:	$\mathsf{ScMul}_{\mathsf{base}}(g, y)$:	$\mathsf{ScMulVf}_{\mathsf{base}}(g, y)$:	$\mathsf{ScSam}_p()$:
return	return g^y	if $g \notin \mathcal{G}$: return $I_{\mathcal{G}}$	$y \leftarrow_{\mathrm{R}} 1 \dots p$
$\mathsf{Map2Pt}(\mathsf{H}_1(\mathsf{pw} \| \mathsf{oc}(P_i, P_j)))$		else: return g^y	return y

Figure 6: Protocol $\mathrm{CPace}_{\mathsf{1MAP}}$ for an elliptic curve group $\mathcal{G}$ of prime order p, over finite field $\mathbb{F}_q$. Generators are computed as $\mathsf{Map2Pt}(\mathsf{H}_1(\mathsf{pw}))$ with a hash function $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{F}_q$. Differences to $\mathrm{CPace}_{\mathsf{base}}$ are marked gray.

6 Analysis of Real-World CPace

The currently most efficient way to run CPace is over elliptic curves. Therefore, from this point onwards, we consider $\mathcal{G}$ to be an elliptic curve constructed over field $\mathbb{F}_q$. From a historical perspective, both CPace research and implementation first focused on prime order curves, such as the NIST-P-256 curve [18]. Subsequently significantly improved performance was shown on Montgomery- and (twisted-)Edwards curves, notably Curve25519 and Ed448 curves [10, 29], which both have a small cofactor c in their group order $c \cdot p$. These approaches consider also implementation pitfalls, e.g., by designing the curve such that there are no incentives for implementers to use insecure speed-ups. Thirdly, recently ideal group abstractions have been presented in order to avoid the complexity of small cofactors in the group order [28, 17], while maintaining all advantages of curves with cofactor.

For smooth integration into each of these different curve ecosystems, CPace needs to be instantiated slightly differently regarding, e.g., computation of the DH generator, group size, multiplication and sampling algorithms. In this section, we analyze how such differences impact security. Using our modular approach with assumption libraries called by a simulator, we are able to present security in terms of differences from our basic CPace analysis in Section 5 in a concise way.

6.1 CPace without Hashing to the Group

We now analyze a variant of the CPace protocol case-tailored for elliptic curve groups $\mathcal{G}$ over finite field $\mathbb{F}_q$. The protocol is depicted in Fig. 6. The only difference to $\mathrm{CPace}_{\mathsf{base}}$ analyzed in the previous section is how parties compute the generators: now the function H_1 hashes onto the field $\mathbb{F}_q$, and generators are computed as $g \leftarrow \mathsf{Map2Pt}(\mathsf{H}_1(\mathsf{pw}))$ for a function $\mathsf{Map2Pt} : \mathbb{F}_q \to \mathcal{G}$. This way, the H_1 outputs can be considered to form an alternative encoding of group elements, where $\mathsf{Map2Pt}$ decodes to the group. $\mathsf{ScMul}, \mathsf{ScMulVf}$ and SamSc are as in Section 5.

Security analysis. Compared to the analysis of $\mathrm{CPace}_{\mathsf{base}}$, the security analysis is complicated by the different computation of the generators in essentially two ways: first, the possibly non-uniform distribution of $\mathsf{Map2Pt}$ induces non-uniformity of DH generators computed by the parties. Second, embedding of trapdoors no longer works by simply programming elements with known exponents into H_1. Instead, the proof will exploit that $\mathsf{Map2Pt}$ is probabilistically invertible, such that preimages of generators with known exponents can be programmed into H_1 instead. Consequently, security of CPace will be based on the $\mathcal{D}_{\mathcal{G}} - \mathsf{sSDH}$ problem Definition 3.3 instead of the sSDH problem, where the distribution $\mathcal{D}_{\mathcal{G}}$ corresponds to the distribution of group elements $\mathsf{Map2Pt}(h_i)$ obtained for uniformly sampled field elements $h_i \leftarrow_{\mathrm{R}} \mathbb{F}_q$. All these changes can be captured by

```
def class D_G _sSDH:
  def _init_(s, Map2Pt, sSDHExp):
    s.sSDH = sSDHExp; s.records = [];
    s.nmax = n_max; s.preim = Map2Pt.PreImages;
  def sampleY(s): return (s.sSDH).sampleY();
  def isValid(X): return (s.sSDH).isValid(X);
  def sampleH1(s):
    g ← (s.sSDH).sampleH1();
    while (1):
      r ←R F_p; preimageList = (s.preim)(g^r); m ←R {0 ... (s.Map2Pt.n_max − 1)};
      if len(preimageList) > m:
        if r = 0: abort("Sampled neutral element.");
        h ←preimageList[m];if h in s.records: abort("H1 collision");
        s.records.append(r, g^r, h); return h;
  def corrupt(s, h, Y):
    if there is (r, g, h) in s.records: return (s.sSDH).corrupt(g, Y^{1/r});
  def DDH(s,h, Y, X, K):
    if there is (r, g, h) in s.records: return (s.sSDH).DDH(g, Y, X, K^{1/r});
# Chaining the experiments for CPace on prime order curve, full (x,y) coordinates
sSdhExp = sSDH(sCDH);
distExp = D_G _sSDH(Map2Pt,sSdhExp);
```

Figure 7: Experiment class definition $\mathcal{D}_{\mathcal{G}}$-sSDH using single executions of Map2Pt, where H_1 hashes to $\mathbb{F}_q$.

replacing library sSDH with a new library for $\mathcal{D}_{\mathcal{G}} - \mathsf{sSDH}$, as we demonstrate below.

Theorem 6.1 (Security of $\text{CPace}_{\mathsf{1MAP}}$). *Let $\lambda, p, q \in \mathbb{N}$ with p prime. Let $\mathcal{G}$ an elliptic curve of order p over field $\mathbb{F}_q$. Let $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{F}_q, \mathsf{H}_2 : \{0,1\}^* \to \{0,1\}^\lambda$ be two hash functions and Map2Pt : $\mathbb{F}_q \to \mathcal{G}$ probabilistically invertible with bound Map2Pt.n_{max}. If the sCDH and sSDH problems are hard in $\mathcal{G}$, then the CPace protocol depicted in Fig. 2 UC-emulates $\mathcal{F}_{\mathsf{lePAKE}}$ in the random-oracle model with respect to adaptive corruptions and both hash functions modeled as random oracles. More precisely, for every adversary $\mathcal{A}$, there exist adversaries $\mathcal{B}_{\mathsf{sSDH}}$ and $\mathcal{B}_{\mathsf{sCDH}}$ against the sSDH and sCDH problems such that*

$$\begin{aligned} &|Pr[Real(\mathcal{Z}, \mathcal{A}, CPace_{\mathsf{1MAP}})] - Pr[Ideal(\mathcal{F}_{\mathsf{lePAKE}}, \mathcal{S})]| \\ &\leq (\mathsf{Map2Pt}.n_{max}) l_{\mathsf{H}_1}/q + (l_{\mathsf{H}_1})^2/p + (\mathsf{Map2Pt}.n_{max} \cdot l_{\mathsf{H}_1})^2/q \\ &\qquad + 2l_{\mathsf{H}_1}^2 \mathbf{Adv}^{\mathsf{sSDH}}_{\mathcal{B}_{\mathsf{sSDH}}}(\mathcal{G}) + \mathbf{Adv}^{\mathsf{sCDH}}_{\mathcal{B}_{\mathsf{sCDH}}}(\mathcal{G}) \end{aligned}$$

where l_{H_1} denotes the number of H_1 queries made by the adversary $\mathcal{A}$ and the simulator $\mathcal{S}$ is as in Fig. 5 but using the object distExp (cf. Fig. 7) instead of the object sSdhExp.

Proof Sketch. Let $\mathcal{D}_{\mathcal{G}}$ denote the distribution on $\mathcal{G}$ induced by Map2Pt. First note, that if the sSDH is hard in $\mathcal{G}$ then the corresponding $\mathcal{D}_{\mathcal{G}}$-sSDH problem is hard by Theorem 3.6 as $\mathsf{Map2Pt}^{-1}$ (implemented in the body of the sampleH1 method by the distExp object) is a rejection sampler for $\mathcal{D}_{\mathcal{G}}$.

We adjust the simulator for "basic" CPace from Fig. 5 as follows. First, we embed the reduction strategy from Theorem 3.6 into an experiment library that converts sSDH challenges into $\mathcal{D}_{\mathcal{G}}$ − sSDH challenges and obtain the class $\mathcal{D}_{\mathcal{G}}$_sSDH depicted in Fig. 7. The class $\mathcal{D}_{\mathcal{G}}$_sSDH uses the Map2Pt.PreImages function (passed as a constructor parameter) for implementing the $\mathsf{Map2Pt}^{-1}$ as defined in Algorithm 1 and an instance of the sSDH class implementing a sSDH experiment that is assigned to a member variable.

Each time the main body of the simulator from Fig. 5 makes calls to its exp object, the corresponding method of the new $\mathcal{D}_{\mathcal{G}}$_sSDH object will be executed, which itself translates the queries into calls to the sSDH object that was passed as constructor parameter.

Importantly, $\mathcal{D}_{\mathcal{G}}$_sSDH provides the same public API as the sSDH class with the distinction that sampling for H_1 returns results from $\mathbb{F}_q$ instead of $\mathcal{G}$. Moreover $\mathcal{D}_{\mathcal{G}}$_sSDH aborts if the code of its sSDH object aborts and, now additionally, also upon H_1 collisions.

We explain now how the indistinguishability argument of Theorem 5.1 needs to be adjusted in order to work for Theorem 6.1 and this new simulator. The first difference applies in game $\mathbf{G}_2$, where we must make sure that the distribution of points provided by the $\mathcal{D}_{\mathcal{G}}$_sSDH object is uniform in $\mathbb{F}_q$ as was in the previous game. This is the case due to Corollary 3.9. In game $\mathbf{G}_3$ no change is needed except for adjusting the collision probability following the derivation from Corollary 3.10 which is now bound by $(n_{\max} \cdot l_{\mathsf{H}_1})^2/q$ in addition to the previous $l_{\mathsf{H}_1}^2/p$ probability. The probability that sampleH1 aborts because it samples the identity element from the distribution is bounded by $(\mathsf{Map2Pt}.n_{\max})l_{\mathsf{H}_1}/q$. Apart of these modification the proof applies without further changes. □

Instantiating Map2Pt. Various constructions have been presented for mapping field elements to elliptic curve points such as Elligator2 [11], simplified SWU [21] and the Shallue-van de Woestijne method (SvdW) [40] (see also [21] and references therein). When considering short-Weierstrass representations of a curve, the general approach is to first derive a set of candidate values x_l for the x coordinate of a point such that for at least one of these candidates x_l there is a coordinate y_l such that (x_l, y_l) is a point on the curve. Subsequently one point (x_l, y_l) is chosen among the candidates. The property of *probabilistic invertibility* is fulfilled for all of the algorithms mentioned above and those currently suggested in [21]. The most generic of these algorithm, SvdW, works for all elliptic curves, while the simplified SWU and Elligator2 algorithms allow for more efficient implementations given that the curve fulfills some constraints.

All these mappings have a fixed and small bound $n_{\max}$ regarding the number of pre-images and come with a PPT algorithm for calculating all preimages. For instance, Elligator2 [11] comes with a maximum $n_{\max} = 2$ of two pre-images per point and $n_{\max} \leq 4$ for the simplified SWU and SvdW algorithms [21]. For all these algorithms, the most complex substep for determining all preimages is the calculation of a small pre-determined number of square roots and inversions in $\mathbb{F}_q$ which can easily be implemented in polynomial time with less computational complexity than one exponentiation operation.

6.2 Map-twice-and-add constructions.

Some mapping constructions aim at producing more uniform distributions using a map-twice-and-add approach, such as presented in [21] and also adopted by the ristretto25519 and decaf ecosystems [17, 28]. We capture this with protocol $\mathsf{CPace}_{\mathsf{2MAP}}$ depicted in Fig. 8. The protocol computes generators by adding two points generated by Map2Pt. H_1 and the library simulating H_1 produce

$\mathrm{CPace}_{\mathsf{2MAP}} := \mathrm{CPace}[\mathsf{Gen}_{\mathsf{2MAP}}, \mathsf{ScMul}_{\mathsf{base}}, \mathsf{ScMulVf}_{\mathsf{base}}, \mathsf{ScSam}_p]$

$\mathsf{Gen}_{\mathsf{2MAP}}(\mathsf{pw}, P_i, P_j)$:
$(h, h') \leftarrow \mathsf{H}_1(\mathsf{pw}||\mathsf{oc}(P_i, P_j))$
return $\mathsf{Map2Pt}(h)$
$\cdot\ \mathsf{Map2Pt}(h')$

$\mathsf{ScMul}_{\mathsf{base}}(g, y)$:
return g^y

$\mathsf{ScMulVf}_{\mathsf{base}}(g, y)$:
if $g \notin \mathcal{G}$: return $I_{\mathcal{G}}$
else: return g^y

$\mathsf{ScSam}_p()$:
$y \leftarrow_{\mathrm{R}} 1 \ldots p$
return y

Figure 8: Protocol $\mathrm{CPace}_{\mathsf{2MAP}}$, with generators computed by two executions of $\mathsf{Map2Pt}$, using hash function $\mathsf{H}_1 : \{0,1\}^* \rightarrow (\mathbb{F}_q)^2$. Other algorithms remain unchanged.

two elements from $\mathbb{F}_q$ instead of a single one. Again we make use of the property of probabilistic invertibility which guarantees (Corollary 3.9) that algorithm two field elements output by the $\mathsf{sampleH1}()$ function implementing the $\mathsf{Map2Pt}^{-1}$ algorithm are uniformly distributed in $\mathbb{F}_q$.

As shown in [30, 13, 21, 22], for many maps the map-twice-and-add approach of $\mathsf{Gen}_{\mathsf{2MAP}}(\mathsf{pw})$ can be shown to produce a distribution that is indistinguishable from a uniform distribution in $\mathcal{G}$. Under this additional assumption the collision bound $\leq l_{\mathsf{H}_1}^2/p$ is maintained at the expense of doubling the computational complexity for calculating generators.

Apart from the uniformity requirement, security holds under the same assumption set as for $\mathrm{CPace}_{\mathsf{1MAP}}$. Specifically $\mathsf{Map2Pt}$ needs to be probabilistically invertible such that the rejection samplers in Fig. 9 can be implemented and such that the simulated H_1 outputs become uniform in $\mathbb{F}_q \times \mathbb{F}_q$.

Theorem 6.2 (Security of $\mathrm{CPace}_{\mathsf{2MAP}}$)**.** *Let $\lambda, p, q \in \mathbb{N}$ with p prime. Let $\mathcal{G}$ an elliptic curve of order p over field $\mathbb{F}_q$. Let $\mathsf{H}_1 : \{0,1\}^* \rightarrow \mathbb{F}_q \times \mathbb{F}_q$, $\mathsf{H}_2 : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ be two hash functions and $\mathsf{Map2Pt} : \mathbb{F}_q \rightarrow \mathcal{G}$ probabilistically invertible with bound $\mathsf{Map2Pt}.n_{max}$. If the sCDH and sSDH problems are hard in $\mathcal{G}$ and if the output of $\mathsf{Gen}_{\mathsf{2MAP}}(.)$ is indistinguishable from a uniform distribution in $\mathcal{G}$, then the $\mathrm{CPace}_{\mathsf{2MAP}}$ protocol depicted in Fig. 8 UC-emulates $\mathcal{F}_{\mathsf{lePAKE}}$ in the random-oracle model with respect to adaptive corruptions and both hash functions modeled as random oracles. More precisely, for every adversary $\mathcal{A}$, there exist adversaries $\mathcal{B}_{\mathsf{sSDH}}$ and $\mathcal{B}_{\mathsf{sCDH}}$ against the sSDH and sCDH problems such that*

$$\begin{aligned} &|Pr[Real(\mathcal{Z}, \mathcal{A}, CPace_{\mathsf{2MAP}})] - Pr[Ideal(\mathcal{F}_{\mathsf{lePAKE}}, \mathcal{S})]| \\ &\quad \leq l_{\mathsf{H}_1}/p + 2l_{\mathsf{H}_1}^2/p + 2l_{\mathsf{H}_1}^2 \mathbf{Adv}^{\mathsf{sSDH}}_{\mathcal{B}_{\mathsf{sSDH}}}(\mathcal{G}) + \mathbf{Adv}^{\mathsf{sCDH}}_{\mathcal{B}_{\mathsf{sCDH}}}(\mathcal{G}) \end{aligned}$$

where l_{H_1} denotes the number of H_1 queries made by the adversary $\mathcal{A}$ and the simulator $\mathcal{S}$ is as in Fig. 5 using the object coffeeExp (cf. Fig. 9).

Proof Sketch. The same approach as for Theorem 6.1 applies. The difference consists in the fact that now the simulator's experiment library additionally aborts if collisions in the final outputs of the $\mathsf{sampleH1}$ method are detected. In order to show that this collision probability is actually only $l_{\mathsf{H}_1}^2/p$ we employ that the map-twice-and-add strategy was required to produce a distribution that is indistinguishable from the uniform distribution in $\mathcal{G}$. The probability that $\mathsf{sampleH1}$ calculates a $r = 0$ (r_1 accidentally sampled such that $r_0 = -r_1$) is bounded by l_{H_1}/p. □

6.3 Considering curves with small co-factor

In this subsection, we now additionally consider that the elliptic curve group $\mathcal{G}$ can be of order $c \cdot p$ with $c \neq 1$, but where Diffie-Hellman-type assumptions can only assumed to be computationally

```
# using python-style notation with self pointer s
def class mapTwice_sSDH:
  def __init__(s, Map2Pt, sSDHExp):
    s.sSDH = sSDHExp; s.records = [];
    s.preim = Map2Pt.PreImages; s.nmax = Map2Pt.n_max
  def sampleY(s): return (s.sSDH).sampleY();
  def isValid(X): return (s.sSDH).isValid(X);
  def sampleH1(s):
    i ← 0;
    g ← (s.sSDH).sampleH1();
    do while (i < 2):
      sample r_i ←R F_p;
      preimageList = (s.preim)(g^{r_i});
      sample m ←R {0 ... (s.n_max − 1)};
      if len(preimageList) > m:
        h_i ←preimageList[m]; i+ = 1;
    h ← (h_0, h_1); r ← (r_0 + r_1);
    if (·, g^r, ·) in s.records abort("Generator collision");
    if r = 0: abort("Neutral element returned");
    s.records.append(r, g^r, h); return h;
  def corrupt(s, h, Y):
    if there is (r, g, h) in s.records:
      return (s.sSDH).corrupt(g, Y^{1/r});
  def DDH(s,h, Y, X, K):
    if there is (r, g, h) in s.records:
      return (s.sSDH).DDH(g, Y, X, K^{1/r});

# Chaining the experiment objects for ristretto and decaf
sSdh = sSDH(sCDH);
coffeeExp = mapTwice_sSDH(Map2Pt,sSdh);
```

Figure 9: Experiment class definition mapTwice_sSDH where H_1 hashes to $\mathbb{F}_q \times \mathbb{F}_q$.

$\mathrm{CPace}_{\mathsf{co}} := \mathrm{CPace}[\mathsf{Gen}_{\mathsf{1MAP}}, \mathsf{ScMul}_{\mathsf{co}}, \mathsf{ScMulVf}_{\mathsf{co}}, \mathsf{ScSam}_p]$			
$\mathsf{Gen}_{\mathsf{1MAP}}(\mathsf{pw}, P_i, P_j)$:	$\mathsf{ScMul}_{\mathsf{co}}(g, y)$:	$\mathsf{ScMulVf}_{\mathsf{co}}(g, y)$:	$\mathsf{ScSam}_p()$:
return	return $g^{c \cdot y}$	if $g \notin \mathcal{G}$: return $I_{\mathcal{G}}$	$y \leftarrow_{\mathrm{R}} 1 \ldots p$
$\mathsf{Map2Pt}(\mathsf{H}_1(\mathsf{pw} \| \mathsf{oc}(P_i, P_j)))$		else: return $g^{c \cdot y}$	return y

Figure 10: Definition of $\mathrm{CPace}_{\mathsf{co}}$ for curves of order $p \cdot c$. The only difference (marked gray) to $\mathrm{CPace}_{\mathsf{1MAP}}$ is that exponents are always multiplied by the cofactor c.

infeasible in the subgroup of order p, denoted $\mathcal{G}_p$. Consequently, $\mathrm{CPace}_{\mathsf{co}}$ on curves with co-factor $c \neq 1$ requires all secret exponents to be multiples of c. Hence, $\mathrm{CPace}_{\mathsf{co}}$ depicted in Fig. 10 deploys modified algorithms ScMul, ScMulVf.

Theorem 6.3 (Security of $\mathrm{CPace}_{\mathsf{co}}$)**.** *Let $\lambda, p, q, c \in \mathbb{N}$, p, c coprime with p prime. Let $\mathcal{G}$ be an elliptic curve of order $p \cdot c$ over field $\mathbb{F}_q$ and $\mathcal{G}_p \subset \mathcal{G}$ a subgroup of order p. Let $CC_c : (g) \mapsto ((g^c)^{1/c \mod p})$ be a cofactor clearing function for c, $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{F}_q, \mathsf{H}_2 : \{0,1\}^* \to \{0,1\}^\lambda$ be two hash functions and $\mathsf{Map2Pt} : \mathbb{F}_q \to \mathcal{G}$ probabilistically invertible with bound $\mathsf{Map2Pt}.n_{max}$. Let be the chained function $\mathsf{Map2Pt}_{\mathcal{G}_p} := (CC_c \circ \mathsf{Map2Pt})$. Let $\mathcal{D}_{\mathcal{G}_p}$ denote the distribution on $\mathcal{G}_p$ induced by $\mathsf{Map2Pt}_{\mathcal{G}_p}$. If the sCDH and sSDH problems are hard in $\mathcal{G}_p$, then the $\mathcal{D}_{\mathcal{G}_p}$-sSDH problem is hard in $\mathcal{G}_p$ and $\mathrm{CPace}_{\mathsf{co}}$*

```
# using python-style notation with self pointer s
def class cofactorClearer:
  "interfaces S to a prime-order experiment class"
  def _init_(s, c, p, primeOrderExpInstance ,p̄):
    s.c=c; s.i= s.c · integer(1/(s.c^2) mod p);s.it= s.c · integer(1/(s.c^2) mod p̄);
    s.exp = primeOrderExpInstance;
  def sampleY(s): return ((s.exp).sampleY())^{s.c};
  def isValid(X): return (s.exp).isValid(X^{s.i})
  def sampleH1(s): return (s.exp).sampleH1();
  def corrupt(s, h, Y): { return (s.exp).corrupt(h, Y^{s.i}); }
  def DDH(s,g, Y, X, K):
    if X ∈ G:  return (s.exp).DDH(g, Y^{s.i}, X^{s.i}, K^{s.i·s.i})
    if X on twist: return (s.exp).DDH(g, Y^{s.i}, X^{s.it}, K^{s.it·s.it})

sSdhExp = sSDH(sCDH);  ccExp = cofactorClearer(sSdhExp);
ccDistExp = D_G _sSDH(Map2Pt,ccExp);
```

Figure 11: Cofactor-clearer class definition use for elliptic curves of order $p \cdot c$ with a quadratic twist having a subgroup of order $\bar{p}$. Note that the inverses $s.i$ and $s.it$ are constructed such that they are multiples of c.

UC-emulates $\mathcal{F}_{\mathsf{lePAKE}}$ *in the random-oracle model with respect to adaptive corruptions when both hash functions are modeled as random oracles. More precisely, for every adversary* $\mathcal{A}$*, there exist adversaries* $\mathcal{B}_{\mathsf{sCDH}}$ *and* $\mathcal{B}_{\mathsf{sSDH}}$ *against the* sCDH *and* sSDH *problems such that*

$$\begin{aligned} &|Pr[Real(\mathcal{Z}, \mathcal{A}, CPace_{\mathsf{co}})] - Pr[Ideal(\mathcal{F}_{\mathsf{lePAKE}}, \mathcal{S})]| \\ \leq\ &(\mathit{Map2Pt}.n_{max} \cdot c) l_{\mathsf{H}_1}/q + 2l_{\mathsf{H}_1}^2/p + (\mathit{Map2Pt}.n_{max} \cdot c \cdot l_{\mathsf{H}_1})^2/q \\ &+2l_{\mathsf{H}_1}^2 \mathbf{Adv}^{\mathsf{sSDH}}_{\mathcal{B}_{\mathsf{sSDH}}}(\mathcal{G}_p) + \mathbf{Adv}^{\mathsf{sCDH}}_{\mathcal{B}_{\mathsf{sCDH}}}(\mathcal{G}_p) \end{aligned}$$

where l_{H_1} *denotes the number of* H_1 *queries made by the adversary* $\mathcal{A}$ *and the simulator* $\mathcal{S}$ *is as in Fig. 5 but using class ccDistExp (cf. Fig. 11) instead of object sSdhExp.*

Proof Sketch. The full group $\mathcal{G}$ has a point g_1 of order c with $g_1^c = I_{\mathcal{G}}$ where $I_{\mathcal{G}}$ denotes the identity element in $\mathcal{G}$, i.e., there are c low-order points $g_1^i, i \in \{1 \dots c\}$. For any point $Y \in \mathcal{G}$ we can consider the points $Y_i = Y \cdot g_i$ as alternative ambiguous representations of the point $CC_c(Y) \in \mathcal{G}_p$. For any input point $Y \in \mathcal{G}_p$, all these c alternative representations can be easily calculated using group operations and g_i. For any of these c alternative representations of Y at most $\mathsf{Map2Pt}.n_{\max}$ preimages will be returned by $\mathsf{Map2Pt.PreImages}$ since $\mathsf{Map2Pt}$ is probabilistically invertible on $\mathcal{G}$. Correspondingly, the probability of accidently drawing a representation of the identity element needs to be multiplied by c and is now bounded by $(\mathsf{Map2Pt}.n_{\max} \cdot c) l_{\mathsf{H}_1}/q$. If up to $\mathsf{Map2Pt}.n_{\max}$ preimages exist per point on the full curve, the chained function $\mathsf{Map2Pt}_{\mathcal{G}_p}$ is probabilistically invertible also on $\mathcal{G}_p$. Its preimage function $\mathsf{Map2Pt}_{\mathcal{G}_p}.\mathsf{PreImages}$ for $\mathcal{G}_p$ can be defined such that it returns all of the preimages of the c ambiguous representations of an input and the maximum number of preimages $\mathsf{Map2Pt}_{\mathcal{G}_p}.n_{\max}$ is, thus, bounded by $\mathsf{Map2Pt}_{\mathcal{G}_p}.n_{\max} = c \cdot \mathsf{Map2Pt}.n_{\max}$. Since we are able to provide all preimages for $\mathsf{Map2Pt}_{\mathcal{G}_p}$ and a bound for their number is known $\mathsf{Map2Pt}_{\mathcal{G}_p}$ is probabistically invertible. We thus can employ Theorem 3.6 and show that if the sSDH is hard in $\mathcal{G}_p$ then the corresponding $\mathcal{D}_{\mathcal{G}_p}$-sSDH problem is also hard.

As $\mathsf{ScMulVf}_{co}$ and ScMul_{co} use exponents that are a multiples of c they are guaranteed to produce a unique result on $\mathcal{G}_p$ for all of the c ambiguous representations of an input point. The additional factor of c in the exponents is compensated by the simulation by calling an experiment library using

the ccExp class from Fig. 11. [3] The ccExp object forwards queries to a $\mathcal{D}_{\mathcal{G}_p}$_sSDH object such that all inputs to the DDH oracle will be in $\mathcal{G}_p$.

Note that without exponents being a multiple of c, we would have had game $\mathbf{G}_3$ and $\mathbf{G}_4$ distinguishable because without the factor c in the secret scalars of honest parties, Y_a and Y_b could have nontrivial low-order components, which is noticeable and does occur in game $\mathbf{G}_4$. □

6.4 CPace using twist secure curves

For a curve in Weierstrass form constructed over a field $\mathbb{F}_q$, a coordinate $x \in \mathbb{F}_q$ represents either the x coordinate of a point on the curve itself or of a point on a second curve, its so-called quadratic twist $\bar{\mathcal{G}}$. On elliptic curves groups $\mathcal{G}$ that provide a property coined *twist security* in [12], CPace allows for implementations with reduced computational complexity and code size under an additional assumption including the group $\bar{\mathcal{G}}$ on the curve's quadratic twist.

Twist secure curve groups $\mathcal{G}$ are characterized by the fact that both the curve and the twist have small co-factors $c, \bar{c}$ and large prime-order subgroups $\mathcal{G}_p$ and $\bar{\mathcal{G}}_{\bar{p}}$ of orders p and $\bar{p}$. Some curves such as Curve25519 [10], Curve19119 [24] and Ed448-Goldilocks [29] have been specifically tailored for this property.

For Diffie-Hellmann protocols where active adversaries are relevant invalid curve attacks need to be considered. As a result, point verification is a crucial substep in most protocols based on Diffie-Hellman style key agreement. Unfortunately incorrectly implemented point verification might easily remain undetected as interacting honest parties are not affected at all.

In order to make implementations resilient against this common pitfall, the popular X448 and X25519 Diffie-Hellman protocols from RFC7748 and [10] employ a specific property of twist-secure curves for offering security without any point verification. To our best knowledge the assumption has not been formally defined previously. For the sake of security analysis of the corresponding CPace construction we formally define the "twist CDH problem" sTCDH on the two prime-order subgroups on the curve and the twist as follows. (Its defined in its *strong* variant allowing for restricted DDH oracle access).

Definition 6.4 (Strong twist CDH problem (sTCDH))**.** Let $\mathcal{G}_p$ be a first cyclic group of prime order p with a generator g and $(Y = g^y)$ sampled uniformly from $(\mathcal{G}_p \setminus I_{\mathcal{G}_p})$. Let $\bar{\mathcal{G}}_{\bar{p}}$ be a second cyclic group of prime order $\bar{p}$ with $\bar{p} \neq p$. Given g, Y and access to DDH oracle $\mathsf{DDH}(g, Y, \cdot, \cdot)$ in $\mathcal{G}_p$, provide $X, Z \in \bar{\mathcal{G}}_{\bar{p}} \setminus I_{\bar{\mathcal{G}}_{\bar{p}}}$ with $Z = X^y$.

Correspondingly, with $\mathbf{Adv}^{\mathsf{sTCDH}}_{\mathcal{B}_{\mathsf{sTCDH}}}(\mathcal{G}_p, \bar{\mathcal{G}}_{\bar{p}})$ we denote the probability that an adversarial algorithm $\mathcal{B}_{\mathsf{sTCDH}}$ calculates a solution for the sTCDH problem for a single randomly sampled challenge for the groups $\mathcal{G}_p, \bar{\mathcal{G}}_{\bar{p}}$ when given access to the restricted DDH oracle in $\mathcal{G}_p$.

Why hardness of this problem allows for the strategy from [10] regarding omitting point verification in protocols such as X448 and X25519 becomes transparent when considering single-coordinate scalar multiplication algorithms. Single-coordinate scalar multiplication algorithms $Z \leftarrow \mathsf{ScMul}(X, y)$ commonly accept inputs X from both, the curve and its twist and return the result $Z = X^y$ on the same curve as the input X.

Using single-coordinate Diffie-Hellman provides the advantage that the adversary may only insert inputs X taken from either the curve or the twist. I.e. if a honest party publishes a public

[3] Note that this class also accepts points on the quadratic twist, a feature that will become relevant only when considering simplified point verification on twist-secure curves as discussed in the upcoming sections.

key $Y = g^y$ an active adversary can provide a point X either on $\mathcal{G}$ or $\bar{\mathcal{G}}$ and make the honest party calculate a Diffie-Hellman result $Z = \mathsf{ScMul}(X, y) = X^y$ where Z comes from the same group as X.

The conventional strong Diffie-Hellman assumption states that the adversary cannot calculate the honest party result point Z for any given random challenge X from $\mathcal{G}_p$. Hardness of the twist problem sTCDH above implies that the adversary also cannot predict the honest party's result point Z for any chosen *adversial* point X from the twist $\bar{\mathcal{G}}_{\bar{p}}$ group. Note that it is easy for the adversary to predict $Z = Y^x$ for an adversially chosen point $X = g^x$ from the original curve group $\mathcal{G}_p$ by following the protocol for a honest participant. Obviously, in comparison, the task of calculating $Z = X^y$ is harder if X needs to be on the twist.

Obviously the sTCDH problem is easy if the order of the points Y or Z is small. For making sure that Y and Z come from the prime-order subgroups, X448 and X25519 employ a co-factor clearing procedure where secret exponents are chosen to be multiples of the co-factor c.

Corollary 6.5. *Let $\mathcal{G}$ be an elliptic curve of order $c \cdot p$ where c, p coprime with a subgroup $\mathcal{G}_p$ of prime order p over base field F_q. Let $\bar{\mathcal{G}}$ be the quadratic twist of $\mathcal{G}$ of order $\bar{p} \cdot \bar{c}$ with a subgroup $\bar{\mathcal{G}}_{\bar{p}}$ of prime order $\bar{p}$. Let c be equal to $\bar{c}$ or an integer multiple of $\bar{c}$. Then for any integer y and point $X \in (\mathcal{G} \cup \bar{\mathcal{G}})$ it holds that $X^{y \cdot c} \in (\mathcal{G}_p \cup \bar{\mathcal{G}}_{\bar{p}})$.*

If the cofactor c is equal or a multiple of the cofactor $\bar{c}$ of the twist, multiplicative co-factor clearing with c is functional for both, the curve and its twist. I.e. for any exponent being a multiple of c and $X \in (\mathcal{G} \cup \bar{\mathcal{G}})$ then $Z = X^{c \cdot y}$ is guaranteed to be either of large prime order or the neutral element of the curve or the twist. Note that the neutral elements on $\mathcal{G}$ and $\bar{\mathcal{G}}$ in X448 and X25519 share the same binary representations (all bits zero). Here checks for a low-order Z corresponds to just verifying that $Z = X^{c \cdot y}$ differs from the representation of the neutral elements.

It is easy to see, as the orders of the groups differ [10], that there is no homomorphic mapping between the curve and its twist. Finally, hardness of sTCDH relates to the hardness of the DLP. sTCDH is easy if the DLP can be solved for (g, Y).

The twist security property can be employed also in the context of CPace for making implementations faster and more resilient against implementation errors regarding the point verification, specifically if CPace is implemented using scalar multiplication based on single-coordinate Montgomery ladders.

Theorem 6.6 (Security of $\text{CPace}_{\mathsf{twist}}$ on twist-secure curves with co-factor.)**.** *Let $\lambda, c, p, \bar{c}, \bar{p}, q \in \mathbb{N}$ with $p, \bar{p}$ prime. Let $\mathcal{G}$ be the group of points on an elliptic curve over field $\mathbb{F}_q$ and $\bar{\mathcal{G}}$ its quadratic twist. Let $p \cdot c$ be the order of $\mathcal{G}$. Let $\bar{p} \cdot \bar{c}$ be the order of $\bar{\mathcal{G}}$. Let $\mathcal{G}_p$ be a subgroup of $\mathcal{G}$ of order p. Let group $\bar{\mathcal{G}}_{\bar{p}}$ be a subgroup of order $\bar{p}$ of $\bar{\mathcal{G}}$. Let c be equal or an integer multiple of $\bar{c}$. Let $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{F}_q, \mathsf{H}_2 : \{0,1\}^* \to \{0,1\}^\lambda$ be two hash functions and $\mathsf{Map2Pt} : \mathbb{F}_q \to \mathcal{G}$ probabilistically invertible with bound $\mathsf{Map2Pt}.n_{max}$.*

Then the protocol $\text{CPace}_{\mathsf{twist}}$ from Fig. 13 UC-emulates $\mathcal{F}_{\mathsf{lePAKE}}$ in the random oracle model with respect to adaptive corruptions and both hash functions modeled as random oracles. More precisely, for every adversary $\mathcal{A}$, there exist adversaries $\mathcal{B}_{\mathsf{sTCDH}}$, $\mathcal{B}_{\mathsf{sSDH}}$ and $\mathcal{B}_{\mathsf{sCDH}}$ against the sTCDH, sSDH and sCDH problems such that

$$
\begin{aligned}
&|Pr[Real(\mathcal{Z}, \mathcal{A}, CPace_{\mathsf{twist}})] - Pr[Ideal(\mathcal{F}_{\mathsf{lePAKE}}, \mathcal{S})]| \leq \\
&(\mathsf{Map2Pt}.n_{max} \cdot c) l_{\mathsf{H}_1}/q + (l_{\mathsf{H}_1})^2/p + (c \cdot \mathsf{Map2Pt}.n_{max} \cdot l_{\mathsf{H}_1})^2/q \\
&\quad + 2 l_{\mathsf{H}_1}^2 \mathbf{Adv}^{\mathsf{sSDH}}_{\mathcal{B}_{\mathsf{sSDH}}}(\mathcal{G}_p) + \mathbf{Adv}^{\mathsf{sCDH}}_{\mathcal{B}_{\mathsf{sCDH}}}(\mathcal{G}_p) + 2 \cdot \mathbf{Adv}^{\mathsf{sTCDH}}_{\mathcal{B}_{\mathsf{sTCDH}}}(\mathcal{G}_p, \bar{\mathcal{G}}_{\bar{p}})
\end{aligned}
$$

```
def class sCDH_sTCDH ():
  "Accepts X, K inputs from G and the twist Ḡ"
  def _init_(s, g): s.g ← g; s.i ← 0; s.s_1, s.s_2 ← fresh;
  def sampleY(s): if s.i ≤ 2: s.i+ = 1; s.y_i ←_R F_p \ 0; s.Y_i ← (s.g)^{y_i}; return s.Y_i;
  def corrupt(s,Y):
    for 1 ≤ m ≤ s.i):
      if (Y = (s.g)^{s.y_m}):
        x ← s.y_m; s.s_i ← corrupt; return x;
  def DDH(s, B, Y, X, K):
    if (B = s.g) and ({Y, X}={s.Y_1, s.Y_2}) and (s.s_1 = fresh) and (s.s_2 = fresh)
              ... and (K = (s.g)^{s.y_1·s.y_2}):
      abort("K solves sCDH(B, Y_1, Y_2)");
    if (B = s.g) and (Y = s.Y_1):
      if(s.s_1 = fresh) and (X ∈ Ḡ \ I_Ḡ) and (X^{s.y_1} = K):
        abort("X, K solve sTCDH(B, Y_1)");
      return (X^{s.y_1} = K);
    if (B = s.g) and (Y = s.Y_2):
      if(s.s_2 = fresh) and (X ∈ Ḡ \ I_Ḡ) and (X^{s.y_2} = K):
        abort("X, K solve sTCDH(B, Y_2)");
      return (X^{s.y_2} = K);
  def isValid(X): return (X ∈ (G \ I_G)) or (X ∈ (Ḡ \ I_Ḡ);

# Chaining the experiment objects for the case of X25519 and X448
twistSdhExp = sSDH(sCDH_sTCDH);
twistCcExp = cofactorClearer(twistSdhExp,c, p, p̄);
twistDistExp = D_G_sSDH(Map2Pt,twistCcExp);
twistXonlyExp =moduloNegationAdapter(twistDistExp);
```

Figure 12: Class combining sCDH and sTCDH experiments. In addition to the previous challenge generator class for sCDH, point verification now also accepts points from the twist except for the twist's neutral element. Moreover the code for the DDH oracle now also accepts its third and fourth inputs X, K from the twist and aborts in case that (X, K) forms a solution of the sTCDH problem for (g, Y).

$\mathrm{CPace}_{\mathsf{twist}}$:=$\mathrm{CPace}[\mathsf{Gen}_{\mathsf{1MAP}}, \mathsf{ScMul}_{\mathsf{co}}, \mathsf{ScMulVf}_{\mathsf{tw}}, \mathsf{ScSam}_p]$

$\mathsf{Gen}_{\mathsf{1MAP}}(\mathsf{pw}, P_i, P_j)$:	$\mathsf{ScMul}_{\mathsf{co}}(g, y)$:	$\mathsf{ScMulVf}_{\mathsf{tw}}(g, y)$:	$\mathsf{ScSam}_p()$:
return $\mathsf{Map2Pt}($	return $g^{c \cdot y}$	if $g \notin \mathcal{G} \cup \bar{\mathcal{G}}$: return $I_{\mathcal{G}}$	$y \leftarrow_{\mathrm{R}} 1 \ldots p$
$\mathsf{H}_1(\mathsf{pw} \Vert \mathsf{oc}(P_i, P_j)))$		$K \leftarrow g^{c \cdot y}$	return y
		if $K = I_{\bar{\mathcal{G}}}$: $K \leftarrow I_{\mathcal{G}}$	
		return K	

Figure 13: Definition of $\mathrm{CPace}_{\mathsf{twist}}$ for curves $\mathcal{G}$ of order $p \cdot c$ with a quadratic twist $\bar{\mathcal{G}}$. The only difference to $\mathrm{CPace}_{\mathsf{co}}$ is that the scalar multiplication functions accept inputs also from the twist.

where l_{H_1} denotes the number of H_1 queries made by the adversary $\mathcal{A}$ and simulator $\mathcal{S}$ is as in Fig. 5 but using the object twistDistExp (cf. Fig. 12) instead of the object sSdhExp.

Proof. The only difference in the real-world protocol between $\text{CPace}_{\mathsf{twist}}$ and $\text{CPace}_{\mathsf{co}}$ consists in the fact that the latter variant's scalar multiplication and verify function now also accepts inputs from the twist. I.e. differing from $\text{CPace}_{\mathsf{co}}$, $\text{CPace}_{\mathsf{twist}}$ issues a session key without aborting if the output of the $\mathsf{ScMulVf}_{\mathsf{tw}}$ function is not the identity element $I_{\mathcal{G}}$.

The difference for the proof strategy for Theorem 5.1 consists in the fact that we now need to handle events where the adversarial strategy is based on injecting points from the twist.

First note that the twist has cofactor $\bar{c}$ and thus $\bar{c}$ low-order points. As c is an integer multiple of $\bar{c}$ the exponentiation by multiples of c maps all inputs from the full twist curve to the twist's prime-order subgroup by Corollary 6.5.

In the simulation we additionally need to handle the case that the adversary provides an input from the twist. We add an additional game after $\mathbf{G}_0$, where we abort if the adversary queries H_2 for Y, X, K for a honest $Y = g^{c \cdot y}$ calculated from a generator g, an adversarial input X from the twist $\bar{\mathcal{G}}_{\bar{p}}$ such that $K = X^{c \cdot y} \in \bar{\mathcal{G}}_{\bar{p}} \setminus I_{\bar{\mathcal{G}}}$, where y is the private scalar of one of the honest parties.

This change is distinguishable only if $\mathcal{A}$ provided a solution to the sTCDH problem from Definition 6.4 and the advantage of a distinguisher in this additional game is bounded by $\mathbf{Adv}^{\mathsf{sTCDH}}_{\mathcal{B}_{\mathsf{sTCDH}}}(\mathcal{G}_p, \bar{\mathcal{G}}_{\bar{p}})$ for the groups $\mathcal{G}_p$ and $\bar{\mathcal{G}}_{\bar{p}}$. In any other case the honest party will return *ISK* values indistinguishable from random values in both, the real world and the ideal world. As the adversary may choose to attack both honest parties, the advantage has to be multiplied by two. We incorporate this change in the simulator by replacing the sCDH object instance that is embedded by the sSDH experiment class by an instance of the $\mathsf{sCDH\_sTCDH}$ class from Fig. 12. □

6.5 CPace using single-coordinate Diffie-Hellman

Some Diffie-Hellman-based protocols, including CPace, can be implemented also on a group modulo negation, i.a. a group where a group element Y and its inverse Y^{-1} (i.e. the point with $I = Y \cdot Y^{-1}$) are not distinguished and share the same binary representation [4].

An elliptic curve in Weierstrass representation becomes a group modulo negation when only using x-coordinates as representation. We use the notation $\hat{Y}$ for such ambiguous encodings and use $\hat{Y} \leftarrow \mathsf{SC}(Y)$ for a function returning the x-coordinate for a point Y and $(Y^{-1}, Y) \leftarrow \mathsf{RC}(\hat{Y})$ for the inverse operation reconstructing Y and Y^{-1} in an undefined order.

The major advantage of using this type of ambiguous encoding is that it can be helpful in practice for all of the following: reducing code size, reducing public key sizes and network bandwidth, avoiding implementation pitfalls [10] and restricting invalid curve attacks to the curve's quadratic twist. Consequently, many real-world protocols such as TLS only use this single coordinate for deriving their session key, as to give implementers the flexibility to take benefit of the above advantages.

For the purpose of function definitions by chaining, we introduce a function $\mathsf{RSC}(\hat{Y}, x)$ that takes one ambigously encoded group element $\hat{Y}$ in addition to one scalar x, i.e. takes the same operands as ScMul. We define $\mathsf{RSC}(\hat{Y}, x)$ such that it returns a tuple (Y, x) such that $\mathsf{SC}(Y) = \hat{Y}$. With this definition, we can formalize CPace using single-coordinate scalar multiplications with the

[4] Counter-examples for protocols that cannot be instantiated on a group modulo negation and require full group structure are, e.g., TBPEKE [39] and SPAKE2 [5]. The reason is that these protocols require addition of arbitrary points on the group.

```
def class moduloNegationAdapter:
  "uses the strip- and reconstruct functions SC and RC."
  def _init_(s,, baseExperiment):
    s.exp ←baseExperiment;s.records← [];
  def sampleY(s): Y ← ((s.exp).sampleY())^{s.c}; s.records.append(Y); return SC(Y);
  def isValid(X̂):(X_0, X_1)←RC(X̂); return (s.exp).isValid(X_0);
  def sampleH1(s): return (s.exp).sampleH1();
  def corrupt(s, h, Ŷ):
    (Y, Y*) ← RC(Ŷ); if Y* in s.records: Y ← Y*; return (s.exp).corrupt(h, Y);
  def DDH(s,g, Ŷ, X̂, K̂):
    (Y, Y*) ← RC(Ŷ); if Y* in s.records: Y ← Y*;
    (X, X*) ← RC(X̂); (K, K*) ← RC(K̂);
    return (s.exp.DDH(g, Y, X, K)) or (s.exp.DDH(g, Y, X, K*))

# Chaining the experiments for prime order curve, single coordinate, single map
sSdhExp = sSDH(sCDH); distExp = D_G _sSDH(Map2Pt,sSdhExp);
singleCoorExp = moduloNegationAdapter(distExp)
```

Figure 14: Single-coordinate experiment class definition for CPace instantiations on groups modulo negation.

chained functions $\mathsf{ScMul}_{x-only} := (\mathsf{SC} \circ \mathsf{ScMul} \circ \mathsf{RSC})$, $\mathsf{ScMulVf}_{x-only} := (\mathsf{SC} \circ \mathsf{ScMulVf} \circ \mathsf{RSC})$ and $\mathsf{Gen}_{x-only} := \mathsf{SC} \circ \mathsf{Gen}$, such that the ambiguous encodings are used.[5]

Theorem 6.7 (Security of $\mathrm{CPace}_{\mathsf{x-only}}$)**.** *Given a group $\mathcal{G}$, assume CPace[*Gen, *ScMul*, *ScMulVf*, *SamSc]* on $\mathcal{G}$ *can be distinguished from an ideal-world run of $\mathcal{F}_{\mathsf{lePAKE}}$ and $\mathcal{S}$ from Fig. 5 with negligible advantage, where $\mathcal{S}$ embeds an experiment object* exp*. Then CPace[*SC ∘ Gen, SC ∘ *ScMul* ∘ RSC, SC ∘ *ScMulVf* ∘ RSC, *SamSc]* on the corresponding group modulo negation $\hat{\mathcal{G}}$ cannot be distinguished from $\mathcal{F}_{\mathsf{lePAKE}}$ running with a simulator $\hat{\mathcal{S}}$ that is obtained by chaining* exp *with* moduloNegationAdapter*, the adapter class from Fig. 14, and the difference in the distinguishing advantage is bounded by a factor of 2.*

Proof Sketch. First note that the functions SC,RC and RSC that implement conversion between group and group modulo negations are all efficient, as in practice the most complex substep is a square root in $\mathbb{F}_q$. Secondly, CPace does not need a full group structure at any point. Instead, only chained exponentiations are used ($K = Y_a^{y_b} = Y_b^{y_a} = g^{y_a \cdot y_b}$), which can likewise be implemented on a group modulo negation. The protocol's correctness is not affected.

When starting with single-coordinate CPace in the real world, the same proof strategy applies, however the collision probability in game $\mathbf{G}_3$ is increased by a factor of 2. In game $\mathbf{G}_4$ no change is required. Also in games $\mathbf{G}_5$ and $\mathbf{G}_6$ the only difficulty shows up when splitting off the code for the full-group versions of the sSDH and sCDH experiments exp from the simulator code. We cannot embed the experiment object exp as-is but need to reconstruct the sign information in order to serve the API of the full-group experiments. The corresponding strategy is implemented in the moduloNegationAdapter adapter class from Fig. 14. Note that this strategy never aborts itself but its incorporated sSDH or sCDH experiments abort upon DDH queries. As the moduloNegationAdapter adapter queries the DDH at most two times, the loss of $\mathrm{CPace}_{\mathsf{x-only}}$ in comparison to CPace is at most a factor of two. □

[5]Note that this definition obtained from chaining with SC and RSC for the scalar multiplications corresponds exactly to the conventional so-called single-coordinate ladder algorithms.

6.6 Chaining the experiment classes

In Appendix G, we describe 3 CPace implementations which combine the single aspects discussed in the previous sections in various combinations. The experiment that encodes the assumptions that apply for the construction that we recommend for use on twist secure Montgomery curves is specified by the "twistXonlyExp" object from Fig. 12. Correspondingly the assumption set that is necessary for proving the security of CPace on the group abstraction construction from Appendix G for ristretto25519 and decaf448 [28, 17] is defined by the "coffeeExp" object from Fig. 9. The assumption set needed for proofs for our recommended construction with short-Weierstrass curves is specified by the "singleCoorExp" object in figure Fig. 14. A concrete example how this process is carried out is given in Appendix F.

References

[1] Michel Abdalla and Manuel Barbosa. Perfect forward security of SPAKE2. Cryptology ePrint Archive, Report 2019/1194, 2019. https://eprint.iacr.org/2019/1194. 40, 41, 42, 49

[2] Michel Abdalla, Manuel Barbosa, Tatiana Bradley, Stanislaw Jarecki, Jonathan Katz, and Jiayu Xu. Universally composable relaxed password authenticated key exchange. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 278–307. Springer, Heidelberg, August 2020. 2, 3, 4, 5, 6, 7, 9, 40, 41, 42, 49

[3] Michel Abdalla, Manuel Barbosa, Jonathan Katz, Julian Loss, and Jiayu Xu. Algebraic adversaries in the universal composability framework. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021*, LNCS. Springer, Heidelberg, December 2021. 3, 4, 5

[4] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In David Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 143–158. Springer, Heidelberg, April 2001. 8

[5] Michel Abdalla and David Pointcheval. Simple password-based encrypted key exchange protocols. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 191–208. Springer, Heidelberg, February 2005. 2, 29, 40

[6] Boaz Barak, Yehuda Lindell, and Tal Rabin. Protocol initialization for the framework of universal composability. Cryptology ePrint Archive, Report 2004/006, 2004. http://eprint.iacr.org/2004/006. 16

[7] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155. Springer, Heidelberg, May 2000. 7, 40, 42

[8] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *1992 IEEE Symposium on Security and Privacy*, pages 72–84. IEEE Computer Society Press, May 1992. 1, 2

[9] 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, *ISC 2009*, volume 5735 of *LNCS*, pages 33–48. Springer, Heidelberg, September 2009. 2

[10] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Heidelberg, April 2006. 3, 20, 26, 27, 29, 51

[11] 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, *ACM CCS 2013*, pages 967–980. ACM Press, November 2013. 11, 22

[12] 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.yp.to/twist.html. 6, 26

[13] Eric Brier, Jean-Sébastien Coron, Thomas Icart, David Madore, Hugues Randriam, and Mehdi Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 237–254. Springer, Heidelberg, August 2010. 11, 23

[14] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. 7

[15] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 404–421. Springer, Heidelberg, May 2005. 3, 4, 7, 9, 49, 50

[16] Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 265–281. Springer, Heidelberg, August 2003. 16

[17] H. de Valence, J. Grigg, G. Tankersley, F. Valsorda, I. Lovecruft, and M. Hamburg. The ristretto255 and decaf448 groups. Rfc, IRTF, 10 2020. 20, 22, 31, 54

[18] Digital Signature Standard (DSS). National Institute of Standards and Technology (NIST), FIPS PUB 186-4, U.S. Department of Commerce, July 2013. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf. 20, 53

[19] Edward Eaton and Douglas Stebila. The "quantum annoying" property of password-authenticated key exchange protocols. In Jung Hee Cheon and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021*, volume 12841 of *LNCS*, pages 154–173. Springer, Heidelberg, July 2021. 3, 49

[20] Elliptic Curve Cryptography. Federal Office for Information Security (BSI), Technical Guideline BSI TR-03111, Version 2.10, June 2018. 53

[21] A. Faz-Hernandez, S. Scott, N. Sullivan, R. Wahby, and C. Wood. Hashing to elliptic curves, 2019. https://datatracker.ietf.org/doc/draft-irtf-cfrg-hash-to-curve/. 4, 11, 22, 23, 53

[22] Pierre-Alain Fouque and Mehdi Tibouchi. Indifferentiable hashing to Barreto-Naehrig curves. In Alejandro Hevia and Gregory Neven, editors, *LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 1–17. Springer, Heidelberg, October 2012. 23

[23] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In Lance Fortnow and Salil P. Vadhan, editors, *43rd ACM STOC*, pages 99–108. ACM Press, June 2011. 5

[24] Björn Haase and Benoît Labrique. Making password authenticated key exchange suitable for resource-constrained industrial control devices. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 346–364. Springer, Heidelberg, September 2017. 26

[25] Björn Haase and Benoît Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR TCHES*, 2019(2):1–48, 2019. https://tches.iacr.org/index.php/TCHES/article/view/7384. 2, 4, 12, 14

[26] Björn Haase. CPace, a balanced composable PAKE, 2020. https://datatracker.ietf.org/doc/draft-haase-cpace/. 1, 2, 14, 51

[27] Björn Haase and Benoît Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. Cryptology ePrint Archive, Report 2018/286, 2018. https://eprint.iacr.org/2018/286. 1

[28] Mike Hamburg. Decaf: Eliminating cofactors through point compression. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 705–723. Springer, Heidelberg, August 2015. 20, 22, 31, 54

[29] Mike Hamburg. Ed448-goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. http://eprint.iacr.org/2015/625. 20, 26

[30] Mike Hamburg. Indifferentiable hashing from elligator 2. Cryptology ePrint Archive, Report 2020/1513, 2020. https://eprint.iacr.org/2020/1513. 11, 23

[31] Julia Hesse. Review of (security of) remaining candidates. Posting to the CFRG mailing list, 2020. https://mailarchive.ietf.org/arch/msg/cfrg/47pnOSsrVS8uozXbAuM-alEk0-s/. 2, 4

[32] Julia Hesse. Separating symmetric and asymmetric password-authenticated key exchange. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 579–599. Springer, Heidelberg, September 2020. 4, 7, 49

[33] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards curves revisited. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 326–343. Springer, Heidelberg, December 2008. 55

[34] David P. Jablon. Strong password-only authenticated key exchange. *Computer Communication Review*, 26(5):5–26, 1996. 2, 12

The protocol is parametrized by a security parameter λ and operates on a group $\mathcal{G}$ of prime order p, of which membership is efficiently decidable. Let $\mathsf{H}_1 : \{0,1\}^* \to \mathcal{G}$ and $\mathsf{H}_2 : \{0,1\}^* \to \{0,1\}^\lambda$ be two hash functions.

Protocol:

1. When $\mathcal{P}$ receives input $(\mathsf{NewSession}, sid, \mathcal{P}, \mathcal{P}', \mathsf{pw})$, it sets $g \leftarrow \mathsf{H}_1(\mathsf{pw})$. $\mathcal{P}$ then samples $y \leftarrow_{\mathrm{R}} \mathbb{Z}_p^*$ and sets $Y \leftarrow g^y$. $\mathcal{P}$ stores a session record $(sid, \mathcal{P}', y, Y, \mathsf{fresh})$ and sends (sid, Y) to $\mathcal{P}'$.

2. When $\mathcal{P}$ receives message (sid, X) from $\mathcal{P}'$ with $X \in \mathcal{G}$, and $\mathcal{P}$ finds a session record $(sid, \mathcal{P}', y, Y, \mathsf{fresh})$, it sets $K \leftarrow X^y$. If $K = I_{\mathcal{G}}$, then $\mathcal{P}$ aborts. If $K \neq I_{\mathcal{G}}$, $\mathcal{P}$ calculates $ISK \leftarrow \mathsf{H}_2(K || \mathsf{oc}(X, Y))$ and outputs (sid, ISK). In either case $\mathcal{P}$ rewrites the session record from fresh to $\mathsf{completed}$.

Figure 15: UC execution of $\mathrm{CPace}_{\mathsf{base}}$ from Figure 2 (bottom), proven secure in Theorem 5.1.

[35] 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, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 456–486. Springer, Heidelberg, April / May 2018. 4, 7, 49

[36] A. Langley, M. Hamburg, and S. Turner. Elliptic Curves for Security. RFC 7748, IETF, January 2016. 6, 54

[37] Manfred Lochter and Johannes Merkle. Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation. RFC 5639, IETF, March 2010. 53

[38] Advanced security mechanism for machine readable travel documents (extended access control (EAC), password authenticated connection establishment (PACE), and restricted identification (RI)). Federal Office for Information Security (BSI), BSI-TR-03110, Version 2.0, 2008. 2

[39] 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, *ASIACCS 17*, pages 301–312. ACM Press, April 2017. 2, 9, 29

[40] Andrew Shallue and Christiaan E. van de Woestijne. Construction of rational points on elliptic curves over finite fields. In *ANTS*, volume 4076 of *Lecture Notes in Computer Science*, pages 510–524. Springer, 2006. 22

[41] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. http://eprint.iacr.org/2004/332. 19

[42] Björn Tackmann. Updated review of PAKEs. Posting to the CFRG mailing list, 2020. https://mailarchive.ietf.org/arch/msg/cfrg/eo8O6JYPmWY6L9TlcIXStFy5gNQ/. 2, 4

A Proof of Theorem 5.1

We start with the real execution of the CPace protocol with an adversary $\mathcal{A}$, and gradually modify it, ending up with the ideal execution $\mathcal{F}_{\mathsf{lePAKE}}$ with a simulator $\mathcal{S}$. The changes will go unnoticed by an (adaptively corrupting) environment $\mathcal{Z}$ interacting with parties and the adversary. For clarity, we formalize the UC execution of $\mathrm{CPace}_{\mathsf{base}}$ in Fig. 15.

Let $\text{Real}_{\mathcal{Z}}(CPace, \mathcal{A})$ be the event that environment $\mathcal{Z}$ with adversary $\mathcal{A}$ and an execution of CPace outputs 1 and $\text{Ideal}_{\mathcal{Z}}(\mathcal{F}_{\mathsf{lePAKE}}, \mathcal{S})$ be the corresponding event in the ideal execution with functionality $\mathcal{F}_{\mathsf{lePAKE}}$ depicted in Fig. 1.

Game $\mathbf{G}_0$: The real protocol execution. This is the real world in which the adversary interacts with real players and may view, modify and/or drop network messages and adaptively corrupt parties.

$$Pr[\text{Real}_{\mathcal{Z}}(\text{CPace}, \mathcal{A})] = Pr[\mathbf{G}_0]$$

Game $\mathbf{G}_1$: Introducing the simulator. In this game we move the whole execution into one machine and call in the simulator $\mathcal{S}$. Note that this implies that $\mathcal{S}$ implements the random oracles $\mathsf{H}_1, \mathsf{H}_2$ and runs the execution with actual passwords as input. We will change the simulation to work without passwords in the upcoming games. In this game for any new input query s for H_1 (H_2) samples a point on the curve (a random string) respectively. No re-programming operation is yet needed. The changes are only syntactical and thus

$$\Pr\,[\mathbf{G}_0\,] = \Pr\,[\mathbf{G}_1\,]$$

Game $\mathbf{G}_2$: Embedding trapdoors. In this game we start keeping track of secret exponents for generators created from passwords. $\mathcal{S}$ samples a fixed generator $g \leftarrow \mathcal{G}$. For every hash query $\mathsf{H}_1(\mathsf{pw})$, $\mathcal{S}$ samples $r \leftarrow \mathbb{F}_p$, stores $(\mathsf{H}_1, \mathsf{pw}, r, r^{-1}, g^r)$ and replies to the query with g^r. The only difference is that the simulator now keeps track of the secret exponents for H_1 queries and the distributions of this and the previous game are perfectly indistinguishable

$$\Pr\,[\mathbf{G}_1\,] = \Pr\,[\mathbf{G}_2\,]$$

Game $\mathbf{G}_3$: Abort on collisions of the random oracle H_1. The simulator aborts if a collision occurs in H_1, i.e., if $\mathcal{S}$ samples an answer for a fresh H_1 query that he already gave before. Note that without collisions two honest parties will always output matching (respectively differing) session keys if both, passwords and party identifiers, match (respectively differ). Note that authenticating party identifiers in addition to the password is important for fending off relay attacks. As the order of the group is $p \approx 2^\lambda$ and $\mathcal{Z}$ can only make a polynomially bounded number l_{H_1}) of H_1 queries, the probability of aborts is negligible in λ by the birthday bound. Thus this and the previous game are indistinguishable.

$$|\ \Pr\,[\mathbf{G}_2\,] - \Pr\,[\mathbf{G}_3\,]\ | \leq l_{\mathsf{H}_1}^2/p$$

Game $\mathbf{G}_4$: Introduce $\mathcal{F}$ and simulate the protocol messages. In the real world all session keys will be calculated by using H_2 queries. In the ideal world session keys of honest parties will not be generated by H_2 queries but will be provided separately by the ideal functionality (with mechanisms in $\mathcal{F}$ for synchronizing H_2 outputs and session keys in case of successful password guesses). Here we prepare this separation and use programming operations for the H_2 RO for keeping session key outputs consistent with H_2.

Changes to the functionality. In this game we add an ITI $\mathcal{F}$ external to the simulator. $\mathcal{F}$ has all interfaces of $\mathcal{F}_{\mathsf{lePAKE}}$ except that we don't yet limit the number of calls to the $\mathsf{LateTestPwd}$ and

TestPwd queries and make the NewKey interface always relay keys coming from the simulator via NewKey queries. Also in this game we let $\mathcal{F}$ inform the simulator about the clear-text passwords upon NewSession events.

Changes to the simulation. Upon receiving an adversarially generated $Y_a \in \mathcal{G} \setminus I_{\mathcal{G}}$ or $Y_b \in \mathcal{G} \setminus I_{\mathcal{G}}$ aimed at a party $\mathcal{P}$, $\mathcal{S}$ sends $(\mathsf{RegisterTest}, \mathcal{P})$ to $\mathcal{F}$. $\mathcal{S}$ simulates protocol messages $Y_a = g^{z_a}$ and $Y_b = g^{z_b}$ on behalf of honest parties by sampling exponents z_a, z_b uniformly from $\mathbb{Z}^*_{q-1}$. Upon an adaptive corruption query of $\mathcal{Z}$ for a party $\mathcal{P}$ sending Y_a (respectively Y_b), $\mathcal{S}$ obtains $\mathcal{P}', \mathsf{pw}, ISK$ from the ideal functionality, computes secret scalars y_a (respectively y_b) used by the real-world protocol as $y_a = z_a \cdot r^{-1}$ (respectively $y_b = z_b \cdot r^{-1}$), where r^{-1} has been looked up in the record $(\mathsf{H}_1, \mathsf{pw}||\mathsf{oc}(\mathcal{P}, \mathcal{P}'), *, r, r^{-1}, g)$. ($\mathcal{S}$ creates a new such record for $\mathsf{pw}||\mathsf{oc}(\mathcal{P}, \mathcal{P}')$ if none yet exists.) Sim then uses y_a to compute K, program $H_2(K||\mathsf{oc}(Y_a, Y_b)) = ISK$ and reveal $(\mathcal{P}', \mathsf{pw}, y_a, Y_a, K, ISK)$ to $\mathcal{A}$ as the internal state of the corrupted $\mathcal{P}$. As $r \neq 0$ uniformly sampling y_a (y_b) or z_a ($z_b y$) is equivalent. In the following we detail how the simulator produces consistent outputs and answers to H_2 queries. While being straightforward, the presentation is slightly involved since the order of queries and messages impacts the simulation.

When $\mathcal{S}$ needs to issue a key to a party $\mathcal{P}$ using password pw for points Y_a and Y_b, then either Y_a or Y_b will have been generated on behalf of $\mathcal{P}$. $\mathcal{S}$ looks for a record $(\mathsf{H}_1, \mathsf{pw}||\mathsf{oc}(\mathcal{P}, \mathcal{P}'), *, r, r^{-1}, g)$ (and creates a H_1 entry for $\mathsf{pw}||\mathsf{oc}(\mathcal{P}, \mathcal{P}')$ if no such record exists). If Y_a was generated for $\mathcal{P}$, then set $K' = Y_b^{z_a r^{-1}}$, otherwise set $K' = Y_a^{z_b r^{-1}}$.

- *Adjust output of $\mathcal{P}$ to earlier H_2 query:* $\mathcal{S}$ then checks whether there is any record $(\mathsf{H}_2, K||\mathsf{oc}(Y_a, Y_b), ISK)$ such that $K = K'$. Note that in this case it holds that both, $DDH(g^r, Y_a, Y_b, K) = DDH(g, Y_a, Y_b, K^{r^{-1}}) = 1$. In this case $\mathcal{S}$ executes the TestPwd query of $\mathcal{F}$ for pw and subsequently passes ISK to the NewKey query of $\mathcal{F}$. We note that our simulator does not make use of the reply "correct/wrong guess" (since there are no more values of the honest party to simulate after the guess happens), but still needs to issue TestPwd in order to be able to determine the attacked party's output via NewKey in case of a successful guess.
- *Align keys in case of matching passwords:* If no corresponding H_2 record is found and both points Y_a, Y_b were generated by honest parties and the passwords of both parties match and a NewKey query has already been issued to the other party, then $\mathcal{S}$ calls NewKey for $\mathcal{P}$ using the key already passed to $\mathcal{P}'$.
- In any other case there is no output yet to keep consistent with. $\mathcal{S}$ samples a new random key just as for new H_2 queries and passes this key to the NewKey query of $\mathcal{F}$.

Upon $\mathcal{Z}$ querying $\mathsf{H}_2(K||Y_a||Y_b)$ for a yet unqueried input,

- *Query not related to any honest output:* If neither Y_a nor Y_b were generated by an honest party or $Y_a||Y_b \neq (Y_a|| < ||Y_b)$ then sample a new random value and output it as result for H_2.
- *Adjust H_2 query to key of $\mathcal{P}$:* Else if all of (1) $Y_a = g^{z_a}$ was simulated for honest party $\mathcal{P}$ and (2) NewKey was already delivered to $\mathcal{F}$ for $\mathcal{P}$ and (3) there is a record $(\mathsf{H}_1, \mathsf{pw}||\mathsf{oc}(\mathcal{P}, \mathcal{P}'), r, r^{-1}, g)$ such that $K = Y_b^{z_a r^{-1}}$ (which occurs if $DDH(g^r, Y_a, Y_b, K) = DDH(g, Y_a, Y_b, K^{r^{-1}}) = 1$) then: $\mathcal{S}$ sends $(\mathsf{LateTestPwd}, \mathcal{P}, \mathsf{pw})$ to $\mathcal{F}$. $\mathcal{S}$ programs $\mathcal{F}$'s answer to this query as reply to the H_2 query.

- *Adjust* H_2 *query to key of* $\mathcal{P}'$*:* Else if $Y_b = g^{z_b}$ was simulated for honest party $\mathcal{P}'$. $\mathcal{S}$ proceeds as for honest $\mathcal{P}$ but checks for $K = Y_a^{z_b r^{-1}}$.
- Else we conclude that there is no need for adjustment and the H_2 query is answered as in the previous game.

Note that with these changes, $\mathcal{S}$ never needs to actually calculate itself a Diffie-Hellman result point K, instead it only makes sure that H_2 queries for parties that adhere to the protocol and keys output to honest parties match. Instead from this point on, the simulators will only need access to $DDH(g, Y, \cdot, \cdot)$ or $DDH(g^r, Y, \cdot, \cdot)$ oracles for a fixed generator g and a honestly generated point Y as second parameter which we could implement easily in this game as we have access to the secret exponents.

Indistinguishability argument. There is no change from the viewpoint of $\mathcal{Z}$ between this and the previous game. The game only differs in the way how session keys and H_2 queries are output. In both games, both, the output of H_2 queries and session keys, are uniformly sampled. Just as in the previous game, session keys output to two honest parties match if $\mathcal{P}$ and $\mathcal{P}'$ have used same password and differ otherwise.

Finally the session keys output to honest parties with password pw match H_2 queries taken for queries using a Diffie-Hellman point K that would be calculated by parties that follow the protocol. It follows that

$$\Pr\left[\mathbf{G}_3\right] = \Pr\left[\mathbf{G}_4\right]$$

Game $\mathbf{G}_5$: Limit number of password guesses

Changes to the functionality. The number of password guesses is now limited to one guess per party. I.e. there is one guess per record $(sid, \mathcal{P}, \mathcal{P}', \mathsf{pw})$.

Changes to the simulation. $\mathcal{S}$ looks for CDH tuples in H_2 queries w.r.t all recorded exponents r in its H_1 list. If $\mathcal{S}$ finds a CDH tuple (G, X, Y, K) that meets the conditions of $\mathbf{G}_4$ (where Y denotes the adversarially-generated message), $\mathcal{S}$ creates a record $(\mathtt{guess}, G, Y)$. If at this point there is already a record $(\mathtt{guess}, G', Y)$ with $G \neq G'$, we say that event `multguess` happens and let $\mathcal{S}$ abort.

Indistinguishability argument. The change is only recognizable if $\mathcal{S}$ has to abort. We show that this happens only with negligible probability if the sSDH assumption holds in $\mathcal{G}$.

We construct an efficient sSDH adversary $\mathcal{B}_{\mathsf{sSDH}}$ interacting with $\mathcal{Z}$. Let (Y, G_1, G_2) denote a sSDH challenge. $\mathcal{B}_{\mathsf{sSDH}}$ embeds the challenge in this game as follows: first, $\mathcal{B}_{\mathsf{sSDH}}$ flips a coin and sets Y to be either the message of $\mathcal{P}$ or $\mathcal{P}'$. $\mathcal{B}_{\mathsf{sSDH}}$ aborts if the chosen party is corrupted or gets corrupted at a later stage. Then, $\mathcal{B}_{\mathsf{sSDH}}$ randomly chooses two out of all H_1 queries made by $\mathcal{Z}$ and answer them with $G_1 \leftarrow \mathsf{H}_1(\mathsf{pw}||\mathsf{oc}(\mathcal{P}, \mathcal{P}'))$ and $G_2 \leftarrow \mathsf{H}_1(\mathsf{pw}'||\mathsf{oc}(\mathcal{P}, \mathcal{P}'))$ (pw and pw' are going to be $\mathcal{Z}$'s two password guesses). $\mathcal{B}_{\mathsf{sSDH}}$ replaces the check performed by the simulator with oracle queries $DDH(G_i, Y, Y', K)$, $i = 1, 2$, where Y' denotes the (simulated or adversarial) other message. If `multguess` occurs, then two CDH solutions were found, namely one for each G_i. $\mathcal{B}_{\mathsf{sSDH}}$ outputs these two solutions. Since w.l.o.g $\mathcal{Z}$ corrupts at most one party and has a view independent of the coin flipped by $\mathcal{B}_{\mathsf{sSDH}}$, $\mathcal{B}_{\mathsf{sSDH}}$ has to abort only with probability $1/2$. Overall, it follows that $Pr[\mathtt{multguess}] \leq 2l^2_{\mathsf{H}_1}\mathbf{Adv}^{\mathsf{sSDH}}_{\mathcal{B}_{\mathsf{sSDH}}}(\mathcal{G})$, where l_{H_1} denotes the number of H_1 queries made by $\mathcal{Z}$. We thus have

$$| \Pr[\mathbf{G}_4] - \Pr[\mathbf{G}_5] | \leq 2l_{\mathsf{H}_1}^2 \mathbf{Adv}^{\mathsf{sSDH}}_{\mathcal{B}_{\mathsf{sSDH}}}(\mathcal{G}).$$

Remark. We note that any reduction could make use of the true password of $\mathcal{P}'$, since a reduction interfaces with $\mathcal{Z}$ and thus also receives protocol inputs. However, it is unclear how to leverage this, since `multguess` implies that $\mathcal{Z}$ makes at least one incorrect guess. Thus, $\mathcal{S}$ needs to turn password guesses into CDH solutions *regardless* of whether a guess is correct or not. Further, we note that a reduction to the strong CDH assumption seems infeasible here. The reason is that $\mathcal{S}$ needs to detect password guesses in H_2 queries, which requires knowledge of the exponent of the simulated message Y_b. The reduction however does not have this knowledge due to Y_b being set to the CDH challenge, and thus needs to leverage DDH oracles w.r.t different generators for detecting both guesses.

Game $\mathbf{G}_6$: Random key if passwords mismatch

Changes to the functionality. In case a record is $\mathsf{interrupted}$, $\mathcal{F}$ now outputs a random key (instead of the one given by the simulator via NewKey).

Changes to the simulation. -

Indistinguishability argument. Assume the output was generated for an honest $\mathcal{P}$ (the other case works analogously). The output towards $\mathcal{Z}$ differs only in case $\mathsf{TestPwd}$ returns "wrong guess" or $\mathsf{LateTestPwd}$ returns a random key. $\mathcal{S}$ only issues these queries if he finds a CDH tuple in H_2. In case of $\mathsf{TestPwd}$ (Y_b was sent after H_2 query), let Y_a, Y_b denote the transcript and K the Diffie-Hellman value computed by $\mathcal{P}$. The environment never submits $(K||\mathsf{oc}(Y_a, Y_b))$ to H_2, as otherwise the previous game would abort due to `multguess` happening. Thus, the output $\mathsf{H}_2(K||\mathsf{oc}(Y_a, Y_b))$ of $\mathcal{P}$ in $\mathbf{G}_5$ is uniformly random from the viewpoint of $\mathcal{Z}$, and replacing it with a fresh random output chosen by $\mathcal{F}$ due to the interrupted record is perfectly indistinguishable.

In case of $\mathsf{LateTestPwd}$ (H_2 was queried after $\mathcal{P}$ generated output), the key output to $\mathcal{P}$ was the randomly chosen K generated by the $\mathsf{LateTestPwd}$ interface of $\mathcal{F}_{\mathsf{lePAKE}}$ in the previous game, which is perfectly indistinguishable from the randomly chosen one that the NewKey interface of $\mathcal{F}_{\mathsf{lePAKE}}$ outputs directly to $\mathcal{P}$ in this game due to the record being interrupted.

Game $\mathbf{G}_7$: Output random keys for honest sessions

In this game, we let the functionality generate parties' outputs in honest sessions. We change the simulation to work without passwords and without knowledge of the honest parties outputs.

Changes to the ideal functionality. We now add the full NewKey interface to $\mathcal{F}$.

Changes to the simulation. Let m denote the number of H_1 queries issued by $\mathcal{Z}$ and $r_1, \ldots r_n \leftarrow \mathbb{F}_q$ the trapdoors embedded in these queries (see $\mathbf{G}_2$). Let z_a, z_b denote the exponents of simulated messages as of $\mathbf{G}_4$. In case $\mathcal{Z}$ queries $\mathsf{H}_2(K||\mathsf{oc}(Y_a, Y_b))$, where for some $i \in [m]$ (g, Y_a, Y_b, K^{r_i}) is a CDH tuple (which can be checked by $\mathcal{S}$ via $g^{z_a z_b} = K^{r_i}$), $\mathcal{S}$ aborts.

Indistinguishability argument. First note that $\mathcal{Z}$ can only note a difference between this and the previous game if it reproduces any value K or K' computed by some (honest) party in a "fresh" and honest session. In particular, this means that $\mathcal{Z}$ cannot corrupt a party nor inject messages (as in this case $\mathcal{S}$ issues $\mathsf{TestPwd}$ or $\mathsf{LateTestPwd}$ and records get interrupted or compromised). Let us detail what party $\mathcal{P}$ computes (argument for $\mathcal{P}'$ is analogously).

W.l.o.g we assume that $\mathcal{Z}$ queried $\mathsf{H}_1(\mathsf{pw}||\mathsf{oc}(\mathcal{P},\mathcal{P}'))$, where pw is the password of $\mathcal{P}$, and obtained g^a as answer. $\mathcal{P}$ was simulated with values g, y_a, and we now implicitly adjust this to $g^a, y_a \cdot 1/a$. This lets $\mathcal{P}$ compute $K \leftarrow Y_b^{y_a 1/a} = g^{y_b y_a 1/a}$, where (g, Y_a, Y_b, K^a) is a CDH tuple. We stress that P computes the same value K regardless of whether both parties use matching or mismatching passwords, since $\mathcal{P}$'s output only depends on the simulated Y_b and is independent of the generator used by $\mathcal{P}'$.

We show that if sCDH holds in $\mathcal{G}$ then $\mathcal{S}$ never aborts. Consider the following efficient adversary $\mathcal{B}_{\mathsf{sCDH}}$. $\mathcal{B}_{\mathsf{sCDH}}$ obtains an sCDH challenge (g, Y_a, Y_b) and executes the simulation of game $\mathbf{G}_7$ with it. Upon Z querying $\mathsf{H}_2(K||\mathsf{oc}(Y_a, Y_b))$, $\mathcal{B}_{\mathsf{sCDH}}$ needs to detect CDH solutions using his own oracle instead of knowledge of exponents of Y_a, Y_b. For this, $\mathcal{B}_{\mathsf{sCDH}}$ calls $b \leftarrow \mathsf{DDH}(g, Y_a, Y_b, K^{a_i})$ and outputs K^{a_i} as sCDH solution if $b = 1$ happens. It follows that

$$| \Pr[\mathbf{G}_6] - \Pr[\mathbf{G}_7] | \leq \mathbf{Adv}^{\mathsf{sCDH}}_{\mathcal{B}_{\mathsf{sCDH}}}(\mathcal{G}).$$

Game $\mathbf{G}_8$: Remove passwords from simulation.

Changes to the ideal functionality. We remove passwords from $\mathsf{NewSession}$ queries sent from $\mathcal{F}$ to $\mathcal{S}$.

Changes to the simulation. As the simulation already is independent of the password, there are no further changes required.

Since we are just removing unused values from the output of $\mathcal{F}$ towards $\mathcal{S}$, the output distributions towards $\mathcal{Z}$ of this and the previous game are indistinguishable. Hence,

$$Pr[\mathbf{G}_7] = Pr[\mathbf{G}_8] = Pr[\mathrm{Ideal}_{\mathcal{Z}}(\mathcal{F}_{\mathsf{lePAKE}}, \mathcal{S})]$$

This game is identical to the ideal execution since $\mathcal{F} = \mathcal{F}_{\mathsf{lePAKE}}$, which concludes the proof. The simulator of this final game is depicted in Fig. 3

B Game-based security analysis

In the main body of this paper we demonstrated strong composability guarantees for several CPace variants. These strong guarantees come at the cost of requiring pre-established unique session identifiers *sid*, for each exchanged key. This might require an additional round of messages in practise, and it is thus a reasonable question to ask which security guarantees CPace features if such overhead is inacceptable for the application. In this section we therefore complement the conducted analysis in the UC framework with a corresponding game-based analysis, exemplarily considering the case of $\mathrm{CPace}_{\mathsf{1MAP}}$. In Fig. 16 we show "sid-free" protocols $\mathrm{CPace}_{\mathsf{1MAP\_NoSID}}$ and $\mathrm{CPace}_{\mathsf{1MAP\_NoSID\_ID}}$.

- **$\mathrm{CPace}_{\mathsf{1MAP\_NoSID}}$** is a symmetric protocol where both parties initially send their messages, and compute their outputs after receiving the message from the peer. Both parties know the identities (P_a, P_b) involved in the exchange, and hence there is no need to inform the other peer about the own identity.

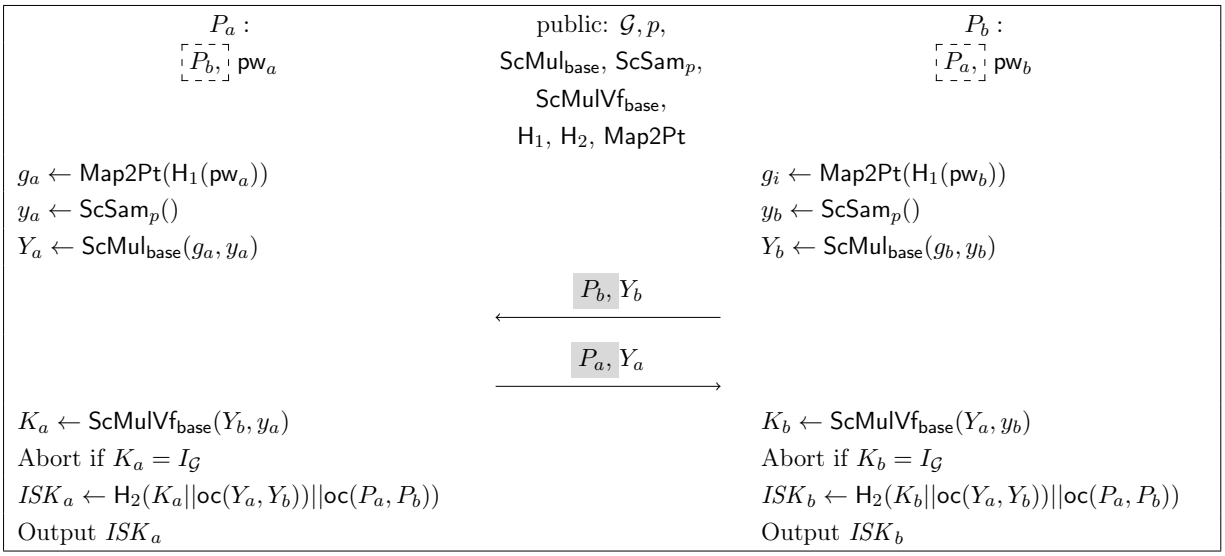

Figure 16: Protocols $\mathsf{CPace}_{\mathsf{1MAP\_NoSID}}$ and $\mathsf{CPace}_{\mathsf{1MAP\_NoSID\_ID}}$ (with initiator P_b and responder P_a), where dashed inputs are only available in $\mathsf{CPace}_{\mathsf{1MAP\_NoSID}}$, and gray message parts only appear in $\mathsf{CPace}_{\mathsf{1MAP\_NoSID\_ID}}$. $\mathsf{ScMul}_{\mathsf{base}}, \mathsf{ScMulVf}_{\mathsf{base}}, \mathsf{ScSam}_p$ are detailed in Figure Fig. 2, and $\mathsf{Map2Pt} : \mathbb{F}_q \to \mathcal{G}$ is probabilistically invertible. With $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{F}_q$ and $\mathsf{H}_2 : \{0,1\}^* \to \{0,1\}^\lambda$ we denote hash functions.

- **$\mathsf{CPace}_{\mathsf{1MAP\_NoSID\_ID}}$** is an asymmetric protocol where one party, say, P_b initializes the key exchange with peer P_a by sending the first message. This message contains identity P_b. P_a only acts upon receipt of such message, informing P_b about its own identity by appending it to the message. Such a protocol layout is useful in cases where P_a might not know under which name (e.g., IP address and port) it got contacted by P_b - might it be because of proxying, NATing, or simply because the API running the PAKE protocol does not have access to the routing information.

B.1 Security Model

We base our analysis on the PAKE security model of Bellare, Pointcheval and Rogaway [7], refined for analysing the PAKE protocol SPAKE2 by Abdalla et al. [1, 2]. The definition comprises specifications of procedures to respond to various queries issued by an adversarial algorithm $\mathcal{A}$. A "real-or-random" game is run with an adversary $\mathcal{A}$ as follows. First INITIALIZE runs and its outputs, except for secrets such as the challenge bit, are passed to $\mathcal{A}$. Subsequently $\mathcal{A}$ is excecuted and its oracle queries are answered by the procedures of the game. When $\mathcal{A}$ terminates, its output is passed to FINALIZE, which returns true if the adversary manages to guess the challenge bit correctly. During the game, $\mathcal{A}$ has access to test queries (we allow multiple such queries, as suggested by Abdalla, Fouque and Pointcheval [5]), which reveal real or random keys according to the challenge bit. A REVEAL query models leakage of session keys, where the adversary obtains the session key computed by an honest party. We assign honest parties a number i, a distinct party identifier bit string P_i and consider that party P_i may be interacting in multiple protocol instances l_i. We use the notation (P_i, l_i) for referring to the protocol instance l_i of party P_i. Initially, at the INITIALIZE operation for

each pair of parties a password is drawn uniformly from a dictionary $\mathcal{D}_{pw}$. We call two instances *partnered* if they share a common communication transcript. Each instance (P_i, l_i) may be in state waiting, accepted or aborted. State waiting corresponds to an instance which has produced its own CPace message but not yet received the remote side's message. State accepted models an instance which has received and processed the remote side's protocol message and calculated its session key ISK. State aborted models an instance of a party which has aborted due to invalid inputs. TEST queries are only allowed if no REVEAL query has been asked for the party itself or a partnered party. The adversary has the option of either querying REVEAL or TEST, but not both.

Adaptions to the security model of Abdalla et al. [1, 2]. Previous game-based security analysis of PAKE protocols did assume protocols which clearly assign initiator and responder roles. Instead, here we also consider the option of a symmetric CPace$_{\mathsf{1MAP_NoSID}}$ construction where any honest party first outputs its own message before processing the remote party's message.

For this, we strenghten the adversary model by allowing $\mathcal{A}$ to adaptively react to the messages sent by honest parties. As a result of this adaptions, we do not have an additional SENDRESP query [1, 2]. The parts in the security games for the initiator-responder instantiation that are not present in the symmetric variant are highlighted by using gray-shaded boxes. In both variants we use the ordered concatenation operation oc as in the symmetric setting there is no natural ordering enforced by the message sequence (see also the corresponding discussion in Appendix D).

Altogether, we give $\mathcal{A}$ access to the following queries:

- $\mathsf{H}_1(pw)$, a hash function taking a bit string and returing an element in $\mathbb{F}_q$. This models that $\mathcal{A}$ has access to the same random oracle as the honest parties and is able to calculate CPace's password-dependent generators as $g_{pw} \leftarrow \mathsf{Map2Pt}(\mathsf{H}_1(pw))$.
- $\mathsf{H}_2(\cdot)$, the hash function used for generating CPace's session key ISK, is a function that takes a bit string of arbitrary length and returing a bit string of size λ.
- SENDINIT(P_i, l_i, P_j) a query returning the message produced by instance l_i of a honest party P_i that was generated for initiating a communication with a remote party P_j. This query leaves (P_i, l_i) in state waiting.
- SENDFINALIZE$(P_j, l_j, (P_i, Y))$, makes a waiting instance (P_j, l_j) transition to state accepted. It models the action the protocol party takes upon reception of the remote side's message. SENDFINALIZE in conjunction with SENDINIT can be used for modeling both, active and passive adversaries.
- REVEAL(P_j, l_j), this query returns $\perp$ if the instance was not accepted or TEST was previously called. Otherwise it marks the transcript of the instance as revealed and reveals the session key.
- TEST(P_i, l_i), if instance l_i of party P_i is not in accepted state or a REVEAL or TEST query has been run previously on an instance with the same transcript it returns $\perp$. Otherwise this query reveals either the session key of P_i, l_i (case $b = 0$) or a freshly sampled random key (case $b = 1$).

Note that we do not have the EXEC(P_i, l_i, P_j, l_j) query from [1]. The reason is that it can be implemented by first calling SENDINIT(P_i, l_i) and SENDINIT(P_j, l_j) and passing the respective

results without any modification to two SENDFINALIZE queries to (P_i, l_i) and (P_j, l_j). I.e., there is no need for a dedicated separate EXEC query returning the transcript that a passive adversary records when eavesdropping the communication between two honest parties as also observed in [7]. We call a query SENDFINALIZE$(P_j, l_j, (P_i, Y))$ to be a *passive attack query* if Y was returned as the result of a previous SENDINIT(P_i, l_i, P_j) query. Otherwise we define a SENDFINALIZE query to be an *active attack query*.

In Fig. 17 we depict how the game is played for CPace$_{\text{1MAP_NoSID}}$ and CPace$_{\text{1MAP_NoSID_ID}}$, where all adversarial queries are implemented by an object of class GameA. (Note that we follow the convention of the Python programming language and use a prepended underscore for distinguishing private functions accessed only from within the class from the public API that is available for the adversary.)

B.2 Game-based security of CPace

Theorem B.1 (Security of CPace$_{\text{1MAP_NoSID}}$ and CPace$_{\text{1MAP_NoSID_ID}}$). *Let $\lambda, p, q \in \mathbb{N}$ with p prime. Let $\mathcal{G}$ be an elliptic curve group of order p over field $\mathbb{F}_q$. Let $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{F}_q, \mathsf{H}_2 : \{0,1\}^* \to \{0,1\}^\lambda$ be two hash functions and $\mathsf{Map2Pt} : \mathbb{F}_q \to \mathcal{G}$ probabilistically invertible with bound $\mathsf{Map2Pt}.n_{max}$. Let $\mathcal{D}_{pw}$ denote a uniformly distributed dictionary of passwords of size $|\mathcal{D}_{pw}|$. If the sCDH and sSDH problems are hard in $\mathcal{G}$, then CPace$_{\text{1MAP_NoSID}}$ (CPace$_{\text{1MAP_NoSID_ID}}$) is secure in the model of Fig. 17, when both hash functions are modeled as random oracles.*

More precisely, for any adversary $\mathcal{A}$ against CPace$_{\text{1MAP_NoSID}}$ (CPace$_{\text{1MAP_NoSID_ID}}$) there exist adversaries $\mathcal{B}_{\mathsf{sCDH}}$ and $\mathcal{B}_{\mathsf{sSDH}}$ against the sCDH and sSDH problems such that the advantage of $\mathcal{A}$ of producing the correct test bit in the FINALIZE query of the security game is bound by

$$\begin{aligned}\mathbf{Adv}(\mathcal{A}) \leq n_{aa}/|\mathcal{D}_{pw}| + \ & (\mathsf{Map2Pt}.n_{max})n_{\mathsf{H}_1}/q + (\mathsf{Map2Pt}.n_{max} \cdot n_{\mathsf{H}_1})^2/q + n_{\mathsf{H}_2}^2/2^\lambda \\ & + 2n_{\mathsf{H}_1}^2 \mathbf{Adv}^{\mathsf{sSDH}}_{\mathcal{B}_{\mathsf{sSDH}}}(\mathcal{G}) + 2\mathbf{Adv}^{\mathsf{sCDH}}_{\mathcal{B}_{\mathsf{sCDH}}}(\mathcal{G}) + (n_s + n_{\mathsf{H}_2})^2/p\end{aligned}$$

where n_{aa} denotes the number of SENDFINALIZE queries for an active attack, n_s denotes the number of SENDINIT queries and n_{H_1} and n_{H_2} denote the number of H_1 and H_2 queries made by $\mathcal{A}$.

For the proof we follow the proof of SPAKE2 [1, 2], using a sequence of game hops. We start with Game$_A$ which represents the security Game for CPace$_{\text{1MAP_NoSID}}$(CPace$_{\text{1MAP_NoSID_ID}}$). Throughout the proof we will modify the security games such that they are indistinguishable from the viewpoint of the adversary except for some bad events, for which the games abort and where we declare the adversary to win.

Game$_B$: Unique transcripts.

In this game we consider collisions in the transcripts and abort if we observe a collision, declaring the adversary to win. We also abort if we observe an accidental collision between a public key Y generated by a SENDINIT query and a preceeding H_2 query which includes the same value Y.[6]

[6] We included this additional abort case in order to be able to disregard a corner case in the upcoming games. With this abort we will be able to disregard the case that such an collision in a H_2 query preceeding the SENDINIT query actually corresponds to some password guess for an instance which point Y that was not yet existing at the time of the H_2 query.

```
# using python-style notation with self pointer s
def class GameA: # Security Game for CPace1MAP_NoSID (CPace1MAP_NoSID_ID)
  def INITIALIZE(s, partyList): # Setup security Game for the set of parties PartyList.
    s.b = SampleFrom({0,1}); s.P = [ ]; # sample the global Real-Or-Random bit b.
    for P_i in partyList:
      s.P.append(P_i);
      for P_j in partyList and not yet in s.P:
        s.pw[P_i, P_j] ←R D_pw # sample password from dictionary that P_i will use with P_j
    s.records = { }; s.tableH1 = { }; s.tableH2 = { }; s.testedTranscripts = []; s.revealedTranscripts = [];
  def FINALIZE(s, b'): return b' == s.b
  def H1(s, str):
    if not str in s.tableH1: s.tableH1[str] = SampleFrom(F_q);
    return s.tableH1[str];
  def _H2(s, str): # "private" method only for calls from within the class itself
    if not str in s.tableH2: s.tableH2[str] = SampleFrom({0,1}^λ);
    return s.tableH2[str];
  def H2(s, str): # API used by the adversary
    return s._H2(str); # give the adversary the same result as for internal hash queries.
  def SENDINIT(s, P_i, l_i, P_j):
    if exists instance (P_i, l_i) in s.records: return ⊥
    inst = { } # setup python dictionary for record entry for new instance
    pw = s.pw[P_i, P_j]; inst["pw"] = pw; inst["Remote"] = P_j; inst["State"] = waiting;
    g = Map2Pt(s.H1(inst["pw"])); inst["g"] = g;
    y = SampleFrom(F_p); inst["y"] = y; Y = g^y; inst["Y"] = Y; s.records [P_i, l_i] = inst;
    return ( P_i, Y); # return first protocol message generated for (P_j, l_i)
  def SENDFINALIZE(s, P_i, l_i, P̂_j, X): # Handle received remote message X from P_j
    if not exists instance (P_i, l_i) in s.records: return ⊥
    inst = s.records [P_i, l_i]; y = inst["y"]; P_j = inst["Remote"]; if(P_j ≠ P̂_j) : inst["State"]=aborted; return;
    inst["K"] = X^y; inst["RemoteY"] = X; inst["Transcr"] = oc(inst["Y"], X) ||oc(P_i, P_j);
    inst["ISK"] = s._H2(inst["K"] || inst["Transcr"]);
    if IsInvalid(X^y): inst["State"] = aborted; else: inst["State"] = accepted
    s.records [P_i, l_i] = inst; # update record.
  def REVEAL(s, P_i, l_i):
    if not exists instance (P_i, l_i) in s.records: return ⊥
    inst = s.records [P_i, l_i];
    if (not inst["State"] == accepted) or (inst["Transcr"] in s.testedTranscripts): return ⊥
    s.revealedTranscripts.append(inst["Transcr"]); return inst["ISK"];
  def TEST(s, P_i, l_i): # Real-or-Random test query
    if not exists instance (P_i, l_i) in s.records: return ⊥
    inst = s.records [P_i, l_i]; transcr = inst["Transcr"];
    if (not inst["State"] == accepted) or (transcr in s.revealedTranscripts ∪ s.testedTranscripts): return ⊥
    s.testedTranscripts.append(transcr);
    if s.b : return SampleRandomKey()
    else: return inst["ISK"];
```

Figure 17: Security game for $\text{CPace}_{\text{1MAP\_NoSID}}$ and $(\text{CPace}_{\text{1MAP\_NoSID\_ID}})$.

As each session transcript with one honest party involved includes an ephemeral public key Y which is uniformly distributed in $\mathcal{G}$, this can be bounded by a statistical analysis to $((n_s + n_{\mathsf{H}_2})^2/p)$ where n is the number of SENDINIT queries and p the order of $\mathcal{G}$.

```
def class GameB(GameA): # Inherit all code from GameA except for overriden methods
  def SENDINIT(s, P_i, l_i, P_j):
    (P_j, Y)= GameA.SENDINIT(P_i, l_i, P_j)
    if Y as substring in s.tableH2: abort()
    return (P_i, Y);
  def SENDFINALIZE(s, P_i, l_i, (P_j, X)):
    GameA.SENDFINALIZE(s, P_i, l_i, (P_j, X)) # First call implementation from the superclass
    if MoreThanTwoSessionInstanceRecordsWithIdenticalTranscripts(s.records): abort()
```

Figure 18: Guarantee unique transcripts.

Game$_C$: Embed H_1 trapdoor.

In this game we replace the implementation of H_1 by an implementation based on Algorithm 1 for $\mathsf{Map2Pt}^{-1}$. As by Corollary 3.9 the modified sampling strategy using $\mathsf{Map2Pt}^{-1}$ still results in uniformly distributed outputs in $\mathbb{F}_q$. This makes the two games indistinguishable. Moreover the adversarial advantage does not differ in comparison to the previous game.

```
def class GameC(GameB): # Inherit all code from GameB except for overriden or extended methods
  def INITIALIZE(s, P):
    GameB.INITIALIZE(P) # First call implementation from the superclass
    s.g = SampleFrom(G \ I_G) # Additionally sample a generator from G
  def H_1(s, str):
    if not str in s.tableH1:
      while True:
        r = SampleFrom(F_p); R = s.g^r
        inverse = Map2Pt^-1(R);
        if inverse ≠⊥:
          s.tableH1[str] = (r, R,inverse); break;
    (r, R, inverse) = s.tableH1[str]
    return inverse;
  def _getGeneratorExponentForH_1(s, str):
    s.H_1(str) # make sure that a table entry for str exists.
    (r, R, inverse) = s.tableH1[str]
    return r;
```

Figure 19: Embed secret exponent trapdoor.

Game$_D$: Abort on hash collisions.

In this game we firstly abort on hash collisions for H_1 and H_2. Regarding H_1 we consider a query $\mathsf{H}_1(str)$ to be colliding if $\mathsf{Map2Pt}(\mathsf{H}_1(str)) = \mathsf{Map2Pt}(\mathsf{H}_1(str'))$ for a previously queried value str'.

The advantage difference for the adversary to the previous game can be bound by statistical analysis to $(n_{\mathsf{H}_1} \cdot \mathsf{Map2Pt}.n_{\max})^2/p + n_{\mathsf{H}_2}^2/2^\lambda$ where n_{H_1} is the number of H_1 queries and n_{H_2} is the number of H_2 queries.

We also abort if for a query str $\mathsf{Map2Pt}(\mathsf{H}_1(str))$ is the neutral element. The probability for accidently drawing the neutral element through H_1 is bound by $(n_{\mathsf{H}_1} \cdot \mathsf{Map2Pt}.n_{\max})/p$.

```
def class GameD(GameC): # Inherit all code from GameC except for overriden or extended methods
  def H1(s, str):
    hashResult = GameC.H1(str)
    if CollisionOfSecretExponentsForDistinctInputs(s.tableH1): abort();
    if SecretExponentZeroForSomeInput(s.tableH1): abort();
    return hashResult;
  def _H2(s, str):
    hashResult = GameC._H2(str)
    if CollisionOfResultsForDistinctInputs(s.tableH2): abort();
    return hashResult;
```

Figure 20: Abort on hash collisions.

Game$_E$: Do not use the password for calculating the SENDINIT result.

In this game we restructure the code such that we do not use the password in SENDINIT queries for calculating the public result message. This game is obviously indistinguishable from the previous one as the obtained distribution for the private exponents y is not modified.

In this game we also defined a private method _DDH that implements a restricted decisional Diffie-Hellman oracle that works for base points derived by the H_1 function and points Y produced by SENDINIT queries. As such the code of the SENDINIT and H_1 queries in conjunction with the _DDH function can be considered to produce challenges for the sCDH and sSDH problems.

```
def class GameE(GameD): # Inherit all code from GameD except for overriden or extended methods
  def SENDINIT(s, P_i, l_i, P_j):
    if exists instance (P_i, l_i) in s.records: return ⊥
    inst = { } # setup python dictionary for record entry for new instance
    pw = s.pw[P_i, P_j]; inst["pw"] = pw; inst["Remote"] = P_j; inst["State"] = waiting;
    z = SampleFrom(F_p); inst["z"] = z;
    Y = s.g^z; inst["Y"] = Y; # calculate Y independently from the password from the global generator s.g .
    result = ( P_i, Y);

    # Calculate y for upcoming SENDFINALIZE calls. Note that inst["y"] will not be used by H_2 code.
    inst["y"] = z / s._getGeneratorExponentForH_1(inst["pw"]);
    s.records [P_i, l_i] = inst;
    return (result); # return first protocol message generated for (P_j, l_i)
  def _DDH(s, g, X, Y, K): # prepare a restricted DDH function using the trapdoor for use in later games.
    Lookup (r, R, inverse) in s.tableH1 such that g == R. If no such record found return ⊥;
    inst = LookupInstanceRecordForPartyWithPublicPoint Y
    if inst ≠⊥:
      z = inst["z"]; return K^(inverse^2) == X^(inverse·z);
    inst = LookupInstanceRecordForPartyWithPublicPoint X
    if inst ≠⊥:
      z = inst["z"]; return K^(inverse^2) == Y^(inverse·z);
    return ⊥ # not a restricted DDH query with valid inputs.
```

Figure 21: Calculate the point Y independently from the passwords in SENDINIT.

Game$_F$: Limit adversaries to one password guess.

We now start giving the adversary and the game-class code different implementations of the H_2 queries.

Firstly note that we previously have ruled out that the adversary has queried H_2 for a string that contains a public key Y that is later returned as the result of a SENDINIT call.

For H_2 queries after a SENDINIT call, we can easily parse the adversary's H_2 hash queries for the transcripts containing public points Y_a and Y_b and the adversarial group element guesses K. If there is any protocol instance (P_i, l_i) which shares it's generated public key with the adversarial query, we can check whether the query for K corresponds to a password-derived generator for some previously queried password pw: For this, we iterate through the H_1 table and use the trapdoor access to the secret exponents r. Also we have access to the secret exponent y used for generating the point Y produced for instance (P_i, l_i). I.e. just as in the case of the UC proof, we can use the secret exponents for implementing a restricted sSDH DDH oracle. This allows us to detect specific values K in the hash query that correspond to password guesses. This way, we are able to count the number of successful adversarial guesses for K for different passwords. If we observe more than a single successful guess, then we let the code abort. If we observe a first password guess by the adversary, we return the adversary the same result as produced by the private $\_\mathsf{H}_2$ query, as in the previous game. However, if we do not detect any password guess, we give the adversary a result from a private random oracle.

In both, the previous game and this game any session key calculated by a protocol instance will necessarily correspond to *some* password. A hash query on the $\_\mathsf{H}_2$ as used by the simulated instances is, thus always unrelated to a H_2 query that corresponds to no password guess. As a result the change in this game can only be distinguished if the adversary's query corresponds to a password guess. Indeed for the first observed password guess the public H_2 is returing the same result as the private oracle $\_\mathsf{H}_2$. We identify this condition by using the _DDH function prepared in the previous game.

On the other hand, the bad event of more than a single password guess can be reduced to the sSDH problem with a loss proportional to the square of the number of H_1 queries (See the corresponding game 5 in the UC proof.) As a result, this and the previous game are indistinguishable except for the bad event that a solution to the sSDH challenge that is provided by the code of the H_1 and SENDINIT queries.

```
def class GameF(GameE): # Inherit all code from GameE except for overriden or extended methods
  def INITIALIZE(s, P):
    GameD.INITIALIZE(P) # First call implementation from the superclass
    s.separateH2Table = { }
  def H2(s, str):
    passwordGuessDetected = False;
    (Ya,Yb, adversarialK) = ParseAdversarialQuery (str)
    for all instances (Pi, li) with public key inst["Y"] matching one public key Ya or Yb in transcript:
      inst = s.records [(Pi, li)];
      Y = inst["Y"]; z = inst["z"]; # observe that none of these values relate to the instance's password!
      Pj = inst["Remote"]; X = parseTranscriptForPublicKeyDifferentFromY (transcript,Y);
      for (r, R, inverse) in s.tableH1:
        if s._DDH(R, X, Y, adversarialK):
          passwordGuessDetected = True;
          if exists entry inst["FirstPasswordGuess"] and inst["FirstPasswordGuess"] ≠ R: abort();
          inst["FirstPasswordGuess"] = R
          s.records[Pi, li] = inst; # store password guess.
    if passwordGuessDetected:
      return s._H2(str); # use the internal oracle
    else: # use a separate private random oracle implementation
      if not str in s.separateH2Table: s.separateH2Table [str] = SampleFrom({0,1}^λ);
      return s.separateH2Table[str]; # use separate RO if not related to a password guess.
```

Figure 22: Limit password guesses to one single guess per instance.

Game$_G$: Abort on successful attacks of passive adversaries.

Again we can easily parse the adversary's H_2 hash queries for the transcripts and group elements K. If two identical transcripts exist in session records for (P_i, l_i) and (P_j, l_j) and as transcripts are unique as a result of the previous games, we easily identify the case of a passive attack. In this case that the adversarial query contains the same value K as calculated by a honest party, we abort and declare the adversary to win. In any other case we respond to the adversary with the results of a private random oracle. Just as in the corresponding case (Game 7 in the sequence used for

the UC security analysis), this bad event can be bound to the adversarial advantage of solving the sCDH problem. We can carry out the reduction by using the self-reducability of the sCDH problem. I.e. we embed a sCDH challenge (g, X, Y) in the form X^{ξ} and Y^{η} for some known random values ξ, η into the executions of the parties P_i and P_j and use the constrained DDH oracle for identifying the correct solution K. In fact we have a loss of a factor of two as we need to flip coins which of the challenge points X or Y to embed in the output of a party P_i as we do not assume clearly assigned user and server roles here.[7]

As a result this game change is only distinguishable in case of the bad event that a solution to the sCDH problem is provided.

```
def class GameG(GameF): # Inherit all code from GameF except for overriden or extended methods
  def H2(s, str):
    (transcript, adversarialK) = ParseAdversarialQueryForTranscriptAndK (str)
    if isTranscriptOfPassiveAttack(transcript):
      (X, Y) = extractPublicInstanceKeysFromTranscript(transcript)
      R = getGeneratorUsedForInstances(transcript)
      if s._DDH(R, X, Y,adversarialK): abort();
      if not str in s.separateH2Table: s.separateH2Table [str] = SampleFrom({0,1}^λ);
      return s.separateH2Table[str]; # use separate RO for passive attack queries.
    else: return GameF.H2(str);
```

Figure 23: Abort on successful attacks by passive adversaries.

Final bound for the adversary advantage. Throughout the game sequence the outputs of the REVEAL and TEST queries have not been modified. However the implementation of the H_2 query was modified such that $\mathcal{A}$ obtains independent values from a private random oracle except for the case of a single password guess iff the tested party instance was *actively* attacked. Note also that the H_2 query that was made accessible to the adversary never accesses the instance passwords and only accesses the secret exponents through the restricted internal _DDH function that was introduced in game E.

When executing a TEST query in Game G considering passive protocol executions, the outputs given to the $\mathcal{A}$ are *always* independent fresh random values taken from the private H_2 oracle except for bad events that can be reduced to the sCDH or sSDH problems.

In the case of actively attacked instances, the only situation where the H_2 oracle that $\mathcal{A}$ accesses and the private $\_\mathsf{H}_2$ oracle produce the same result is if a password guess was detected in Game F. Moreover in Game F this event is limited to one single guess only.

As a result the best available adversarial strategy available in Game G is a random single password guess from the dictionary $\mathcal{D}_{pw}$ for any of the n_{aa} active attacks which occurs with probability $n_{aa}/|\mathcal{D}_{pw}|$. Summarizing the distinguisher advantages, we obtain the bounds from Theorem B.1. □

[7] In fact if the same point X will be embedded into both attacked parties P_i and P_j the abort case corresponds to the event of a solution of the Square-Diffie-Hellman problem. For simplification in this already lengthy paper we decided to not introduce and define this additional problem in this paper and included a loss factor of two for the sCDH problem in the theorem statement instead.

Other variants of CPace. This concludes our game-based security analysis of $\mathrm{CPace}_{\mathsf{1MAP\_NoSID}}$ using a hash-then-map-once approach for calculating the generators. We do not identify any obstacle for carrying out a corresponding analysis also for the other variants discussed in the simulation-based security analysis within this paper. For instance when considering $\mathrm{CPace}_{\mathsf{2MAP}}$, only the way the trapdoor for H_1 is implemented in game C needs to be adapted and otherwise the proof can be carried out without any further change.

Quantum Annoyance. A protocol features quantum annoyance if an adversary with access to a quantum computer still needs to solve at least one instances of a computational problem per attack [19]. CPace is prone to offline dictionary attacks when facing such an adversary. To see this, consider an attacker $\mathcal{A}$ sending a point $Y = g_{\mathsf{pw}}^{y}$ to an honest party, with $g_{\mathsf{pw}} \leftarrow \mathsf{Gen}(\mathsf{pw})$. $\mathcal{A}$ computes $K_{\mathsf{pw}} \leftarrow \mathsf{ScMulVf}_{\mathsf{base}}(Y', y)$ for Y' received from the honest party and derives key ISK_{pw} from it. Without engaging in further communication with the honest party, $\mathcal{A}$ can now compute the final key with respect to another password pw' by computing $g_{\mathsf{pw}'} \leftarrow \mathsf{Gen}(\mathsf{pw}')$, solving a DLP to obtain r such that $g_{\mathsf{pw}'}^{r} = g_{\mathsf{pw}}$, and compute $K_{\mathsf{pw}'} \leftarrow K_{\mathsf{pw}}^{r}$ and derive $ISK_{\mathsf{pw}'}$ from it as usual. The adversary could keep going on like this, having to solve one instance of a DLP per password guess. Moreover, if generators depend only on passwords, as is the case in Fig. 16, then solving a single DLP with respect to g_{pw} allows $\mathcal{A}$ to efficiently compute the key corresponding to pw in *all* actively attacked sessions. For a more quantum annoying CPace, Gen must contain session-specific information. Computing $g_{\mathsf{pw}} \leftarrow \mathsf{Map2Pt}(\mathsf{H}_1(P||P'||\mathsf{pw}))$ forces $\mathcal{A}$ to solve one DLP instance per password *per pair of parties*, and computing $g_{\mathsf{pw}} \leftarrow \mathsf{Map2Pt}(\mathsf{H}_1(sid||P||P'||\mathsf{pw}))$ with session-specific information sid finally forces $\mathcal{A}$ to solve one DLP per password guess.

Conclusion. Our security analysis of CPace in the UC framework in Section 5 requires that, prior to entering the protocol, a pre-established unique session identifier sid is available to both parties. As the above game-based security analysis highlights, CPace features also strong albeit different[8] security guarantees if there is no such *pre*-established sid.

C On shortcomings of UC PAKE functionalities

We provide an illustrating example on how a shortcoming in existing PAKE functionalities from the literature [15, 35, 32, 2] impacts their suitability for building higher-level applications from PAKE. Let us first detail what shortcoming we are talking about. In PAKE functionalities, the adversary $\mathcal{A}$ gets to determine the session key of honest users in some cases. This makes sense if $\mathcal{A}$ manages to guess an honest party's password during an interaction with said party, in which case $\mathcal{A}$ can compute the very same session key and thus the party's output is no longer uniformly random (from the viewpoint of the adversary). But strangely, all existing PAKE functionalities also allow the adversary to determine the honest party's key K^* via NewKey queries if (cf. Fig. 1)

either $\mathcal{P}$ or $\mathcal{P}'$ is corrupted.

[8] While Theorem B.1 does not imply composability and holds only for randomly chosen passwords, it implies weak forward secrecy. Hence, both $\mathrm{CPace}_{\mathsf{1MAP\_NoSID}}$ and $\mathrm{CPace}_{\mathsf{1MAP\_NoSID\_ID}}$ can be turned into PAKEs with perfect forward secrecy by adding key confirmation [1, 2]. We note that, without key confirmation, perfect forward secrecy of $\mathrm{CPace}_{\mathsf{1MAP\_NoSID}}$ and $\mathrm{CPace}_{\mathsf{1MAP\_NoSID\_ID}}$ seems only provable by relying on the algebraic adversary model, a strategy that has been demonstrated to work for SPAKE2 [1, 2].

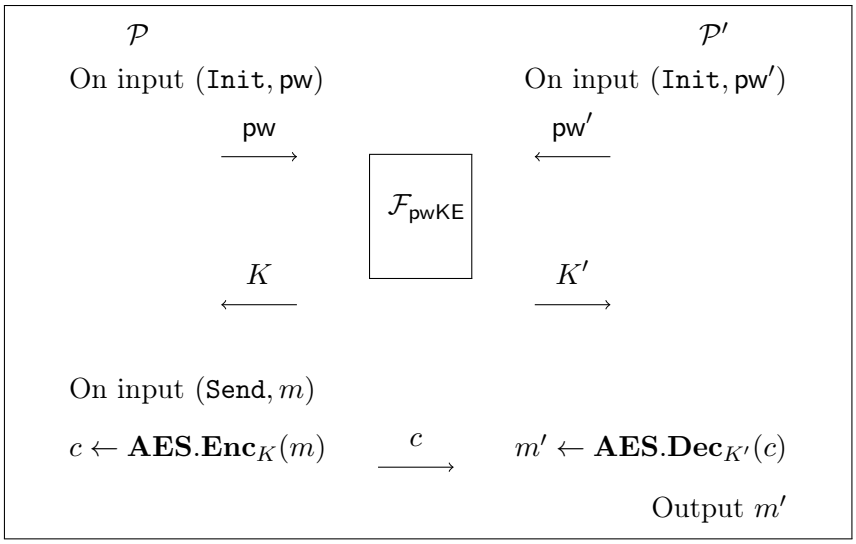

Figure 24: Toy example of a modular password-authenticated secure channel protocol with $\mathcal{F}_{\mathsf{pwKE}}$ as building block, exposing the shortcoming of $\mathcal{F}_{\mathsf{pwKE}}$.

This contradicts the principle of authenticated key exchange, where unauthenticated entities (i.e., honest or malicious parties not knowing the password) should not be allowed to learn the key computed by an honest party.

Does this mean all known PAKE protocols are insecure? Most importantly, all PAKE protocols proven w.r.t any of the existing PAKE functionalities *can still be considered secure*. On a technical level, the reason is that these functionalities still provide all guarantees one would expect from a PAKE against network attackers, as long as no corruptions occur. Besides that, we are not aware of any actual UC PAKE security analysis that exploits the above shortcoming in their simulation, and conjecture that they can be proven secure without this shortcoming.

Why bother then to fix this? The shortcoming's effect shows when a PAKE functionality is used to modularly build other protocols. Modular protocol analysis requires strong composability guarantees of security proofs and is one of the main features of the UC framework. As an example, assume we want to build password-authenticated secure channels from $\mathcal{F}_{\mathsf{pwKE}}$ depicted in Fig. 1. Intuitively, a password-authenticated secure channel allows two parties to securely communicate if *and only if* they hold the same password. Consider the following password-based channel toy protocol[9] Π_{sc} depicted in Fig. 24: users call $\mathcal{F}_{\mathsf{pwKE}}$ to turn their passwords into a cryptographic key and subsequently encrypt a message m under this key using a symmetric cipher.

Intuitively, we would expect protocol Π_{sc} to implement a secure password-authenticated channel. Unfortunately, with the shortcoming in existing PAKE functionalities, there is no way to prove this protocol secure: upon corrupting $\mathcal{P}'$, the adversary gets to determine the value K sent to an honest $\mathcal{P}$. A simulator would now have to produce a ciphertext c that, for any K chosen by the adversary, decrypts to m – but without knowing m. Clearly, this traps the simulator.

Lastly, we note that our argument above is backed up by Canetti et al. [15], who could only circumvent the above simulation trap by integrating the shortcoming also into their password-based

[9]This is just a demonstrating example and not a suggestion for a practical protocol, which would require authenticated encryption [15].

channel functionality.

To summarize, it seems necessary to strengthen UC PAKE functionalities in the way we propose in this paper, in order to make them useful as building blocks for higher level applications.

D Initiator-responder and parallel CPace protocol variants

The current CPace specification in [26] describes a protocol with clear initiator and responder roles, where the responder sends his reply only upon reception of the initiator message. Astonishingly, we observed that for an analysis of such a protocol in the UC framework, an ideal PAKE functionality technically needs a richer structure than the one for a corresponding *parallel* protocol that does not enforce ordering. The reason is that the responder party needs to be activated twice, once for sending the network message and once for issuing the session key.

In order to avoid this purely technical complexity in our presentation we decided to analyze security of CPace here in the more complex setting where no ordering is enforced. For the purpose of the analysis, we thus had to modify the protocol from [26] such that ordered concatenation (written oc(A,B)) is used for generating hash function inputs instead of just putting the initiator's message first.

We would like to stress that in case that the protocol control flow guarantees a defined sequence, as is e.g. the case for protocols such as TLS, there is no need security-wise to enforce use of ordered concatenation.

E Note on sampling of scalars

The CPace protocol needs an algorithm for sampling scalars y for use as private Diffie-Hellman exponents. Ideally for a (sub-)group of order p, these should be sampled from a uniform distribution from $\mathbb{F}_p \setminus 0$. Sometimes, in particular if p is very close to a power of two $p = 2^l + p_0$ with $|p_0| \ll p$, existing Diffie-Hellman libraries draw scalars from the set $\{1, \ldots, (2^l - 1)\}$ instead or include other structure in the scalars. Examples include the X25519 Diffie-Hellman protocol [10] on Curve25519 with it's so-called scalar-clamping procedure which has been re-used for the X448 protocol on Curve448. For the analysis of CPace this change can be considered by a specific scalar sampling function ScSam.

We also observed libraries (for X25519 and X448, but also libraries targeting short-Weierstrass curves) to fix some scalar bits to a defined 1 value in order to guarantee for a constant execution time in window-based scalar multiplication strategy. There are also good reasons for avoiding any additional calculation involving secret scalars y, such as might be required during rejection sampling, since any operation on y might put the secret at risk, e.g. due to side-channels. As CPace aims at being suitable for re-using existing well-tested libraries as-is, the implications for the protocol security due to non-uniform sampling of scalars needs to be analyzed. Unlike for conventional Diffie-Hellman protocols, for CPace not only the confidentiality of ephemeral session keys might be affected but also information on the passwords could be leaked.

In order to consider the case of $p = 2^l + p_0$, we will introduce an additional game $\mathbf{G}_0{}'$ after the real-world setting and only change the scalar-sampling algorithm. In this game, we replace the possibly slightly non-uniform distribution of secret scalars y_a from honest parties in the real world with a fully uniform distribution. For a group order $c \cdot (2^l + p_0)$ implementations in the real world honest parties might draw the scalars y_a and y_b from $\{1, \ldots, (2^l - 1)\}$ instead of $\{1, \ldots, (p - 1)\}$.

Let $\mathbf{Adv}^{\mathrm{DUN}}_{\mathcal{B}_{\mathrm{DUN}}}(\mathcal{G}, \mathsf{ScSam})$ be the probability of an adversarial algorithm $\mathcal{B}_{\mathrm{DUN}}$ of distinguishing a group element $Y \in \mathcal{G}$, where $y \leftarrow \mathsf{ScSam}(), Y = g^y$ was calculated by use of a function ScSam, from a uniformly drawn element $X \leftarrow_{\mathrm{R}} \mathcal{G} \setminus I_{\mathcal{G}}$. Let l_1 denote the number of generated public keys Y_a, Y_b from honest parties and let l_2 the number of adaptive corruptions that reveal the secret exponents. Then the following bound for the possible distinguisher advantage applies:

$$| \Pr[\mathbf{G}_0] - \Pr[\mathbf{G}_0{}'] | \leq |p_0/p| \cdot (l_1 \cdot \mathbf{Adv}^{\mathrm{DUN}}_{\mathcal{B}_{\mathrm{DUN}}}(\mathcal{G}, \mathsf{ScSam}) + l_2)$$

As a result of our assessment the structure introduced by the "scalar clamping" defined for X25519 and X448 does not introduce a critical non-uniformity for CPace.

F Details on chaining challenge generator classes

Two different types of experiments can be distinguished. Firstly, the sCDH and the sCDH_sTCDH classes (Fig. 5 and Fig. 12) share the same API and specify two variants of the conventional CDH problem. Instances will be assigned a base point g in their constructor call, produce two elements from a prime-order group and answer DDH queries. Secret exponents will be revealed in case of corruption events. Also the objects inform the caller on which objects are accepted by their DDH oracle.

This API is used by the objects representing variants of the simultaneous Diffie-Hellman assumption. As security of all variants is ultimately based on the prime-order version of the sSDH problem all calls from $\mathcal{S}$ will ultimately be converted into a call to an instance of the sSDH class from Fig. 4. This is implemented by chaining the different classes.

As root of the chain first an instance of the prime-order sSDH class is generated which is parametrized by one of the two conventional CDH classes (sCDH or sCDH_sTCDH) by a constructor parameter. In its constructor the sSDH object will sample a generator g and create an instance of the sCDH or sCDH_sTCDH class, which will become a member of the sSDH root class. The base point is passed to this member object in its constructor call.

The challenge generator classes for the *simultaneous* problem variants will all produce base points for the simultaneous Diffie-Hellman problem through their API for producing H_1 samples. However the encoding of the base point varies. In the root class sSDH a uniformly sampled group element is returned directly. The derived classes could return the base point also in form of an encoding h that needs to be passed to the mapping construction for decoding.

We describe the chaining approach by using the example of single-coordinate protocol variants on curves with cofactor and twist security using a "map-once" primitive (our recommendation for twist secure Montgomery curves from Appendix G). The corresponding code is found in Fig. 12 below the class definition.

As all Montgomery curves having real-world relevance today come with twist security, the sSDH root object is instantiated with an sCDH_sTCDH object and we obtain an "twistSdhExp" instance of the sSDH class. As the curve has a cofactor, "twistSdhExp" will be passed to an instance "twistCcExp" of the class cofactorClearer and stored there as a member. "twistCcExp" makes sure that inputs for all queries, notably parameters 2,3,4 of the DDH function, will be mapped to the prime-order subgroup and transformed into calls to "twistSdhExp". The "twistCcExp" object itself is then passed to the constructor of the distribution class $\mathcal{D}_{\mathcal{G}}$_sSDH which will be given a function reference to the implementation of the preimage calculator function for the map. This "twistDistExp" object will return H_1 samples that need to be decoded using the Map2Pt function

in order to obtain group elements and require this encoding in its first operand in its DDH function. Finally an instance "twistXonlyExp" of the moduloNegationAdapter adapter class is created, having the "twistDistExp" object as a member in its body.

Calls to the DDH method of the "twistXonlyExp" object will receive the base point h in a form that needs to be passed to Map2Pt for decoding to a group element. The missing coordinate is reconstructed for the second, third and fourth operand of the DDH function and for both candidate points calls to the DDH method of the "twistDistExp" instance will be issued. The latter will convert the base point query h to a group element and forward the query to the cofactor clearer. The latter will make sure that queries for points will have their low-order component cleared and issue a call to the sSDH root object which itself will forward the query to the sCDH_sTCDH instance in its body which uses its secret exponents for giving the response.

G Recommendations: How to instantiate CPace

CPace was designed for suitability with different styles of Diffie-Hellman protocols on elliptic curves. Our perception is that it is possible to distinguish three main application scenarios with real-world relevance.

G.1 Short-Weierstrass

The first elliptic curves being standardized used a short-Weierstrass form. In practice only prime-order curves became of more widespread relevance. While some applications use point-compression, historically mostly a full coordinate representation is used for the network communication, (possibly for patent circumvention reasons that applied at the time). A typical example would be ECC on NIST-P256 and Brainpool curves [18, 20, 37].

In this ecosystem, we recommend to instantiate CPace using the following features

- Encode points Y_a and Y_b using full coordinates and use conventional point verification.
- Still, we recommend to actually only use the x-coordinate of the Diffie-Hellman result K for the session key, as TLS does so and some libraries use an x-coordinate-only ladder internally.
- We recommend to use the *nonuniform* Encode2Curve algorithms from the Hash2Curve draft, i.e. the "map-once" primitive, specifically because otherwise single-coordinate scalar multiplication strategies would not be practical for CPace. The point addition required by the "map-twice-and-add" approach would require full group operations. As mapping algorithm we recommend simplified SWU [21] because according to our analysis this is the least complex and most efficient variant working for the established curve set, notably NIST-P256.
- In this ecosystems, implementers should carefully evaluate the process of scalar sampling chosen in their library, in particular for the NIST-P256 and Brainpool curves. We observed that some libraries might not sample scalars sufficiently uniformly. Here we recommend rejection sampling.

For this instantiation the assumption set is modeled by the challenge-generator class "singleCoorExp" in Fig. 14.

G.2 Montgomery curve ladders

In addition to the established Weierstrass ecosystem, recently constructions based on Montgomery and (twisted) Edwards curves emerged. We believe that these designs became especially attractive as their design already considered typical implementation pitfalls from the very beginning. The wide-spread use of the prominent representatives X25519 and X448 [36] resulted in standardization for internet protocols and by standardization bodies such as NIST.

The first implementations for CPace did focus on this type of Diffie-Hellman primitives. For the twist-secure Curve25519 and Ed449 we recommend the following configuration.

- Only use the u-coordinate on the Montgomery curve.
- Do not verify the curve equation explicitly, but check the neutral elements (all elements encoded with zeros), i.e. use the twist security. This way ScMulVf and ScMul become the same function and X25519(X,y) and X448(X, y) can directly be used for this purpose. Note that checking for the neutral elements (all zero encoding for X448 and X25519) is absolutely mandatory for CPace. Here the discussion from [36] which describes this check as optional does not (!) apply.
- For the map, we recommend to use non-uniform Elligator2 without co-factor clearing as this spares a field inversion (or alternatively spares a montgomery ladder implementation working on projective-coordinate inputs instead of an affine input.).

As the group orders are very close to a power of two, the clamped scalar method can be considered suitable also for CPace instantiations[10] For the construction described in the this paragraph the assumption set is modeled by the challenge-generator class "twistXonlyExp" from Fig. 12.

G.3 Group abstraction

The speed advantage and the simplicity provided by the efficient and complete addition formulas available for Montgomery and (twisted) Edwards curves can only be obtained at the cost that protocols then need to deal with the co-factors. Not all protocols are, such as CPace, analyzed regarding this aspect. In order to allow for "the best of both worlds" projects are emerging which provide prime-order group abstractions on these curves, such as ristretto25519 and decaf448 [28, 17]. As these abstractions work with the most efficient addition formulas currently known for elliptic curves, we do not see any incentive or advantage for single-coordinate instantiations for CPace. (In our opinion tightly constrained applications might rather opt for the Montgomery curve ladders anyway.) The designers of these abstraction frameworks also include mapping algorithms ("batteries included") and enforce invalid-curve checks by their API design. Here we recommend the following parameters for CPace:

- Use the compressed point encoding from the abstraction.
- Use the built-in map-twice-and-add construction that comes "batteries included" with the abstraction.

[10] As the u-coordinate-only approach does not distinguish the curve points g^y and $g^{-y} = g^{p-y}$, also fixing the most significant scalar bit to one is not critical here.

- In our opinion, the verification check for the identity element should receive some attention. Comparisons should be carried out using the unambiguous encoding provided by the abstraction and *not* using the intermediate representation format employed internally for the Hisil-Wong-Carter-Dawson addition formulas [33].

As the group orders for ristretto25519 and decaf448 are very close to a power of two, using random-number generator outputs without rejection sampling can be considered suitable for deriving secret exponents for CPace.

For this instantiation the assumption set is modeled by the challenge-generator class "coffeeExp" in Fig. 9.