Human Computing for Handling Strong Corruptions in Authenticated Key Exchange

Alexandra Boldyreva, Shan Chen, Pierre-Alain Dupont, David Pointcheval

▶ To cite this version:

Alexandra Boldyreva, Shan Chen, Pierre-Alain Dupont, David Pointcheval. Human Computing for Handling Strong Corruptions in Authenticated Key Exchange. CSF 2017 - 30th IEEE Computer Security Foundations Symposium, Aug 2017, Santa Barbara, CA, United States. pp.159 - 175, 10.1109/CSF.2017.31 . hal-01628797

HAL Id: hal-01628797
https://inria.hal.science/hal-01628797v1

Submitted on 6 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A preliminary version of this paper appears in the *30th IEEE Computer Security Foundations Symposium, CSF 2017.*

Human Computing for Handling Strong Corruptions in Authenticated Key Exchange

Alexandra Boldyreva[1], Shan Chen[1], Pierre-Alain Dupont[2,3], and David Pointcheval[2,3]

[1] School of Computer Science, Georgia Institute of Technology
266 Ferst Dr., Atlanta, GA 30332, USA
{sasha,shanchen}@gatech.edu
[2] Département d'informatique de l'ENS, École Normale Supérieure
CNRS, PSL Research University, 75005 Paris, France
[3] INRIA
{pierre-alain.dupont,David.Pointcheval}@ens.fr

Abstract. We propose the first user authentication and key exchange protocols that can tolerate strong corruptions on the client-side. If a user happens to log in to a server from a terminal that has been fully compromised, then the other past and future user's sessions initiated from honest terminals stay secure. We define the security model for Human Authenticated Key Exchange (HAKE) protocols and first propose two generic protocols based on human-compatible (HC) function family, password-authenticated key exchange (PAKE), commitment, and authenticated encryption. We prove our HAKE protocols secure under reasonable assumptions and discuss efficient instantiations. We thereafter propose a variant where the human gets help from a small device such as RSA SecurID. This permits to implement an HC function family with stronger security and thus allows to weaken required assumptions on the PAKE. This leads to the very efficient HAKE which is still secure in case of strong corruptions. We believe that our work will promote further developments in the area of human-oriented cryptography.

1 Introduction

Motivation and Focus. Consider a very common scenario when a user needs to log in to and securely communicate to a server, with which she shares a secret. This problem has been extensively studied under the name of Password-Authenticated Key Exchange (or PAKE), since the seminal paper by Bellovin and Merritt [BM92]. But what happens if the client terminal the user logs in from has been compromised? The machine may have a spyware keylogger recording the user's keystrokes and sending them to the attacker. A session hijacking malware may alter the legitimate computation and impersonate the user or the server.

The existing security definitions for PAKE acknowledge the problem by modeling *strong* corruptions when the adversary learns all the current state of the machine. However, none of the existing protocols try to offer any solution in this case. Basically, the consensus is that in case of strong corruption, all is lost to the user, and the only thing guaranteed is that this should not violate security of other users. Indeed, cryptography cannot do much since the attacker invading a machine would know everything, as it can read all secrets being stored or typed.

In this paper, we take a fresh look at this problem of strong corruptions with the intention of providing a solution. The informal goal is as follows. Given fully untrusted machines, a user's sessions (past and future) from other trusted terminals are still protected, even though the same long-term secret is used. As we said, it seems like nothing can be done cryptographically. But there are possibilities. The basic idea is to store no long-term secrets on the machines, and instead, employ human computation or an additional secure device such as RSA SecurID to boost security. (We think it is much more reasonable to assume that the human or the small device not connected to a network stays uncompromised, than terminals and other devices used for connecting to servers.) In a bit more detail, we ask the human user to log in by computing (in her head or with an additional device) a function of the memorized

long-term secret and a challenge sent by the server, and entering it into the terminal. Then we can use a PAKE-like protocol ran on the response as a common ephemeral secret, also known as a *one-time password*.
A PAKE, that is usually used to prevent off-line dictionary attacks, here provides the guarantee that no information is leaked about the one-time passwords in passive and even active sessions. It is important to limit the information leakage about the long-term secret of the user, since one-time passwords, were they in the clear, could have helped recovering the long-term secret. This is unfortunately the case when they are generated with functions that are easy enough to be computed by a human. On the other hand, if an additional device is used to derive the one-time passwords, their privacy may be less critical, and so resistance to off-line dictionary attacks is not required anymore, which allows the use of a weaker variant of PAKE. To make these ideas "work", numerous problems need to be resolved to finalize the solutions. We discuss these after we describe our security model.

Protocol and Security Definitions. The novelty behind our definitions is the unavoidable incorporation of a human player. We define a *human authenticated key exchange* (or HAKE) protocol as an interactive protocol between a human user U and a server S, via a terminal T. The server can only directly communicate with the terminal, and the user can only directly communicate with the terminal. In addition, the messages sent by the terminal to the user must be *human-readable*, the messages sent by the user to the terminal must be *human-writable*, and the long-term secret of the user must be *human-memorizable*, unless an additional device is used for computing the ephemeral secrets.
Our security model is a non-trivial extension of the security model for PAKE protocols by Bellare, Pointcheval, and Rogaway [BPR00], later called BPR, as we take into account really strong corruptions and model human computations/interactions. As already mentioned, the goal of a HAKE protocol is to ensure that a human user sharing the long-term secret with a server can establish a secure channel with the server, in presence of a very strong attacker. Our model takes into account various types of attacks possible in practice. As usual, to model a network compromise (e.g., taking advantage of an insecure Wi-Fi), we allow the adversary to control the messages parties exchange. The attacker can read and modify the communication between a server and a terminal. However, we assume that the channel between the human user and the *honest* terminal is secure, since this is a direct communication from the keyboard and the screen. At least, it is authentic and private, unless the terminal is compromised. We thus also have to model malicious terminals and this, in fact, is the gist of our work. Even though taking the full control of a computer is an extremely hard job, compromising its parts, such as a browser, is very common. And such compromises can be of various strengths. Using a keylogger, screen capture or similar malware the adversary can learn the terminal's inputs/outputs. The attacker may also learn some random coins or intermediate values from the internal state of the compromised computer. This also models human "over-the-shoulder" attacks. Even though such compromise can be referred to as honest-but-curious, the existing protocols, such as PAKE, do not offer protection against it. The existing protocols only protect against *weak* corruptions, where the attacker just learns the session key (with reveal-queries). This models the misuse of the session key, rather than the terminal compromise. Even if security models for PAKE allow the attacker to learn the long-term secrets [BPR00] (with corrupt-queries), or the internal states (in the UC framework [Can01]), this is only to model forward secrecy, and so the security of past sessions, but nothing is guaranteed anymore for future sessions.
In our model, we let the adversary compromise terminals and learn all their inputs and the internal state. Moreover, we consider an even more powerful adversary, who takes full control of the terminal's browser and can display outputs of its choice to be shown to the human user. Hence, the adversary can interact with the human user, but is never given the long-term secret key memorized by the human user (or stored on her secondary device).
The security goals are, to the most part, the standard privacy and authentication for key exchange protocols: we want to make sure that an attacker cannot learn any information about the session key nor make a party agree on a session key without the other party completing the protocol. Of course, if

a terminal is compromised, it is unreasonable to expect security of the current session. But this should not compromise security of other sessions (past or future), even involving the same user.

HAKE Protocol: Generic Constructions and Instantiations. Let us assume we have a human-compatible function family F (we will discuss it in more detail shortly). Let the server pick a random challenge x (or increment a counter) and display it to the human user via the user's terminal. The user can compute (in her head or using a device) and enter the response $r = F_K(x)$, where K is the long-term secret shared between the user and the server. The server can compute r on its end the same way. Then, the terminal and the server execute a PAKE protocol on r (i.e., the response r plays the role of the password in PAKE), and thus agree on a session key. Even though human-computable responses may have low entropy, PAKE ensures security against off-line dictionary attacks, which guarantees no information leakage about the ephemeral secret r in passive sessions, and even in active sessions, excepted possibly the exclusion of one candidate per session. If the attacker compromises the terminal, a suitable "unforgeability" property of F would prevent the adversary from breaking security of other sessions.
But still, this protocol is not secure under our definition. An attacker, who learns an ephemeral secret $r = F_K(x)$ for a given challenge x can later use it to successfully impersonate the server, by forcing the same challenge. To prevent such replay attacks, we let the terminal and the server to jointly pick a challenge using a coin-flipping protocol, that we implement using a commitment scheme with specific properties. This is our first proposal, which we call the *Basic HAKE*: using a coin-flipping, we avoid replay attacks, and with a suitable unforgeability' property on the function family F we can guarantee the security of the global process.
However, a malicious terminal can still ask specific (not necessarily random) challenges to the human user, while impersonating the server, and the user has no way to detect such a malicious behavior. Therefore security of the Basic HAKE requires that the HC function unforgeability holds even in presence of multiple adaptive challenges. This may be too strong of a requirement in practice. We thereafter enhance *Basic HAKE* and propose the *Confirmed HAKE* protocol, which allows parties to detect potential bad behaviors, in order to react appropriately, and thus the construction tolerates weaker HC function families. Requirements on HC function families then become more compatible with functions that can be evaluated by human being without external help. The Confirmed HAKE also provides explicit authentication of the parties.
Finally, we consider the case of a device-assisted protocol: with such an additional device, one can implement more complex computations, and thus use stronger HC function families. This leads to less critical ephemeral secrets: leaking information about several $(x, F_K(x))$ pairs might not endanger the long-term secret K. This allows us to rely on a weaker variant of PAKE, and hence get a device-assisted HAKE that is more efficient.

Human-Compatible Function Family. We now turn our attention to the inner part of the construction, the human-compatible function family. Security-wise, the adversary should be able to see multiple challenge-response pairs, among which some of the challenges could be chosen by the attacker (adaptive queries vs. non-adaptive queries). This is because the attacker, who compromises a terminal, can eavesdrop on the communication with the human user. And an active attacker who took control of the terminal can impersonate the server and ask the user to answer maliciously chosen challenges. But still, the adversary should not be able to forge a valid response for a new random challenge, so that future sessions remain safe.
Finding such a function family would be easy if we did not have the human-computability restrictions. We survey some works on secure human-based computation later, but they are not directly suitable for us. Luckily, a recent paper "Towards human computable passwords" by Blocki, Blum, Datta, and Vempala [BBDV16] (almost) provides a solution. The paper proposes a way for a human user to authenticate to a computer that does not offer privacy (honest-but-curious). Such a computer stores a set of challenges and the user authenticates by providing a response to a random challenge. In their concrete construction, a challenge is a set of images, the secret user memorizes is a correspondence between images and numbers and the response is some basic function using addition of the digits

(modulo 10). The authors provide experimental evidence that their scheme can be used by a human user. Namely, the secret can be memorized and the response can be computed within reasonable time by an average human user. The authors also propose a tool to help secret memorization. While the usability of their solution is not perfect, it is definitely a start and further research will hopefully yield protocols with better usability.
Security-wise, the authors prove that recovering the user's long-term secret from a number (below a certain bound) of random challenge-response pairs (non-adaptive queries) is equivalent to solving the random planted constraint satisfiability problem, and they state a conjecture about security of the latter. To support the conjecture, the authors prove the hardness of the problem for any *statistical* attacker, extending the results of [FPV15a]. Finally, it is proven that forging a response for a random challenge is equivalent to recovering the secret. The bound on the number of revealed challenge-response pairs corresponds to the maximum number of logins a user can execute, without endangering future sessions.
The construction and security results from [BBDV16] are very useful for our work, but we cannot use them as is. The problem is that it is not known whether security of their scheme holds when the attacker can see responses to maliciously chosen challenges (adaptive queries). We extend their analysis and prove a second conjecture that the unforgeability of their HC function family still holds if the adversary can make very few adaptive queries. Our Confirmed HAKE is designed to rely on such HC functions (whose security can tolerate very few adaptive queries): after the PAKE completion using the first response, the human user selects a random challenge and enters it into the terminal, who encrypts the challenge under the recently established session key and forwards the result to the server. The server decrypts, computes the response, and sends it, also encrypted to the terminal. The terminal decrypts and displays the response and the human user verifies it. If verification fails, the user needs to take measures against suspected terminal infection and possibly abort the long-term secret. Encrypting the terminal-server communication here is needed for authenticity in case of an honest terminal, to prevent a network adversary to ask the server maliciously chosen challenges. We show that this extended protocol limits the number of responses the attacker infecting the terminal can obtain for malicious challenges of its choice (in that case, the adversary will not be able to make the user pass the connection confirmation step). We argue that this addition, while adds a little bit more work for the human user, does not violate human computability for our instantiation, i.e., that the user can select a random challenge and verify the response. Furthermore, we show that the Confirmed HAKE provides explicit authentication assuming that the encryption scheme is secure authenticated encryption.
Our formal analysis on the HC function [BBDV16] demands a stronger conjecture stating *one-more* unforgeability. This is similar to the analysis of blind signatures that relies on the one-more unforgeability of RSA [BNPS03], but we consider a sequential version of the one-more security definition, that is weaker than the original one.
We want to note that, unlike [BBDV16], in our analysis, the bound on the number of challenge-response pairs the attacker can see does not correspond to the total number of logins, but only to the number of logins via compromised terminals, which is much more practical. This is because PAKE guarantees security against network attackers when end-points are secure: responses remain completely hidden to external players.
Unfortunately, it is not clear how to extend the results of [BBDV16] to expect resistance to many adaptive queries (so that we could have a simpler protocol without the confirmation step). The only possibility is the use of a pseudo-random function: after many adaptive queries, the response to a new challenge is still random-looking to any adversary. But for such functions, one needs additional help, hence our *device-assisted* scenario. One important advantage of such a stronger HC function family (tolerating many adaptive challenges, and thus also many non-adaptive challenges) is that responses are ephemeral secrets used once for authentication, but that can be revealed after use: as a consequence, a weaker variant of PAKE is enough, since resistance to off-line dictionary attacks is not required any more. We can expect more efficient constructions. Hence is our first construction in the device-assisted

context. But to limit interactions with the device and avoid collisions on the inputs, we thereafter adopt a time-based challenge: $r = F_K(t)$, with an increasing counter t, based on an internal clock. While one cannot guarantee perfect synchronization between the device and the server, we can tolerate a slight time-shift since we anyway use timeframes that are long enough for the human to enter the response read on the device (e.g., 30 seconds or 1 minute).

Related Work. As we mentioned, there are numerous results about PAKE, from the seminal paper of Bellovin and Merritt [BM92, BPR00], but they offer no practical solutions for strong corruptions, in order to protect future sessions. There are various cryptographic schemes involving human participants, using graphical identification [vABHL03, KI96, KPZ98, JMM+99, DP00, TvO04, BCVO12, WK04], some of them offer security against shoulder surfing. But they offer no security if the terminal is fully compromised.

Matsumoto and Imai [MI91] proposed the first scheme to deal with human identification through insecure channels (and via untrusted machines). The scheme has been improved by the follow up works [WHT95, Mat96, LT99]. However, the schemes are only secure given very few login sessions or require the human to memorize a long bitstring. We view the aforementioned paper by Blocki et al. [BBDV16] as an improvement over the results of this line of work.

Dziembowski [Dzi10] also considers the problem of human-based key-exchange, but in a setting where both parties are human and his scheme is only secure against a machine adversary assumed to be unable to solve CAPTCHAs.

There is a long sequence of papers [JW05, BC08, BCD06, MP07, LMM08, Kho14, Kho15] following the work by Hopper and Blum [HB01] offering protocols for the same problem of secure human identification over insecure channels, whose security is based on the Learning Parity with Noise problem. With error-correcting codes, such protocols could be adapted to generate deterministic responses (which is required by our HC function definition), but usability will not be good for the same reason as most HB-type protocols are not really suitable for humans. Personal devices generating one-time passwords have been commercially available for years [RSA], motivating IETF to standardize their constructions and use in many protocols [Hal95, HMNS98, Nys07, MBH+05, MMPR11]. The work [MMPR11] is particularly relevant as it defined a time-based one-time password algorithm based on HMAC [MBH+05]. Interestingly, while dedicated token generators are the most secure, software applications running on mobile phones are now commonly used [Goo].

Some papers also explore PAKE schemes with one-time passwords. Paterson and Stebila [PS10] define a security model for one-time PAKE, explicitly considering the compromise of past (and future) one-time passwords, but still recovering the security after a compromise, thanks to the ephemeral property of the one-time password and its change over the time. Unfortunately, their construction is a generic one, using a PAKE as a black-box. It thus cannot be more efficient than a PAKE, whereas preventing off-line dictionary attacks is not required in this setting. Our goal is to get a more efficient construction than any PAKE protocol, which we achieve with our device-assisted HAKE in Section 6. The authors of [PS10] mention the possibility of using a secure token to generate the one-time passwords and then running one-time-PAKE on it, but they did not provide an explicit protocol or security analysis.

Open Problems. We hope that our work will stimulate further results about secure human-compatible cryptographic function families. We leave to future works to formally prove the unforgeability property (against several adaptive queries) of the HC function from [BBDV16], and possibly finding other HC function families with such security. Those would allow to avoid additional devices and still have a completely proven efficient HAKE protocol. Improving the usability of the scheme from [BBDV16] will indeed imply improved HAKE protocols, and may have other applications. Another interesting question is to design a coin-flipping protocol with a human participant. Such protocol could be used within HAKE to prevent the attacker to ask malicious challenges. Eventually, after this first step of modeling HAKE protocols with symmetric long-term secrets shared between the user and the server, asymmetric secrets would be important to consider. This would be similar to the so-called *verifier-based PAKE* that helps moderate the impact of corruption of the server.

2 Human Authenticated Key Exchange (HAKE)

2.1 HAKE Definitions

In this section, we define a human authenticated key exchange (HAKE) protocol, as an extension of [BPR00].

Protocol Participants. We fix the set of participants to be $\mathsf{ID} = \{U_\ell\}_\ell \cup \{T\} \cup \{S\}$, which contains finite number of human users U_ℓ, one terminal T and one server S. And we assume that each member is uniquely described by a bitstring. In the real life, each user U_ℓ can communicate with multiple servers via multiple terminals. But we justify below why considering a single terminal and a single server is sufficient.

Human-Compatible Communication. Here we present several notions that our protocol definition will use. Since it is hard to formalize human computational abilities, our definitions are not mathematically precise.
We say a message is *human-readable* if this is a short sequence of ASCII symbols, or images; *human-writable* if this is a short sequence of ASCII symbols[1]; *human-memorizable* if this is simple enough to be memorized by an average human, e.g., a simple arithmetic rule like "plus 3 modulo 10". A function is *human-computable* if an average human can evaluate it without help of additional resources other than his head, e.g., simple additions modulo 10. A set is *human-sampleable* if an average human can choose a message from the set at random according to the appropriate distribution without help of additional resources other than his head.

HAKE Syntax. We now formally describe a HAKE protocol.

Definition 1 (HAKE Protocol). *A* human authenticated key exchange *protocol is an interactive protocol between a human user denoted $U \in \{U_\ell\}_\ell$ and the server S, via the terminal T. It consists of two algorithms:*

- *A long-term key generation algorithm $\mathcal{LKG}$ which takes as input the security parameter and outputs a long-term key.*
- *An interactive key-exchange algorithm $\mathcal{KE}$ which is ran between U, T, and S. At the beginning, only U and S take as input the same long-term secret key and, at the end, T and S each outputs a session key sk_T and sk_S respectively. In case of additional explicit authentication, U and/or S may either* accept *or* reject *the connection.*

The above algorithms must satisfy the following constraints:

- *S can only communicate with T;*
- *U can only communicate with T, and*
  - *the message sent by T to U must be* human-readable*, and*
  - *the message sent by U to T must be* human-writable*;*
- *The long term secret and the state of U, if any, must be* human-memorizable *for the duration necessary.*

The **correctness** condition requires that for every security parameter and for every long-term key output by $\mathcal{LKG}$, in any execution of $\mathcal{KE}$, U and S both accept the connection (in case of explicit authentication), T and S complete the protocol with the same session key ($\mathsf{sk}_T = \mathsf{sk}_S$).

2.2 Formal Security Model

In this section, we formally define the security model for a HAKE protocol, which is part of our main contributions. As already mentioned, the goal of a HAKE protocol is to ensure that a human user

[1] It is also possible to incorporate mouse clicks into that, but we do not deal with it for simplicity.

sharing the long-term secret with a server can help a terminal to establish a secure channel with the server, in presence of a very powerful attacker, including strong corruptions of terminals.
As usual, to model multiple and possibly concurrent (except for the human users) sessions we consider oracles π_P^j, where $j \in \mathbb{N}$ and $P \in \mathsf{ID}$. For human oracles, sessions can only be sequential, and not concurrent, meaning that humans are not allowed to run several sessions concurrently (a new session starts after the previous one ends). This is a reasonable assumption for human users. We note that since terminals do not store long-term secrets and do not preserve state between sessions, multiple terminal oracles model both multiple sessions ran from the same or different terminals.
Hence, in the following, we will consider several human users U_ℓ with different long-term secret keys, one terminal T, and one server S, with all the users' long-term secret keys. For all of them, multiple instances will model the multiple sessions (either sequential for U_ℓ, or possibly concurrent for T and S). However, while the server can concurrently run several sessions, we will also limit it to one session at a time with each user: the server will not start a new session with a user until it finishes the previous session with the same user.
Because of our specific context with a human user, there is a direct communication link between the user and the terminal, and so we can assume that the channels between instances $\pi_{U_\ell}^i$ and π_T^j are authenticated and even private (unless the terminal oracle is *compromised*, as defined below), whereas the communication between the terminal and the server is over the internet, and so the channels between instances π_T^j and π_S^k are neither authenticated nor private.

Security Experiments. We consider the following security experiments associated with a given HAKE protocol and an adversary $\mathcal{A}$, to define the two classical security notions for authenticated key exchange: privacy (or semantic security of the session key) and authentication. In these experiments, the adversary $\mathcal{A}$ can make the following queries:

- $\mathtt{Compromise}(j, \ell)$, where $j, \ell \in \mathbb{N}$ – As the result of this query, the terminal-oracle π_T^j is considered to be *compromised*, and the adversary gets its internal state, i.e. the random tape, temporary variables, etc. If the terminal-oracle π_T^j is not linked yet to a user, it is linked to user U_ℓ with the user oracle $\pi_{U_\ell}^i$ for a new index i, otherwise ℓ is ignored;
- $\mathtt{Infect}(j)$, where $j \in \mathbb{N}$ – As the result of this query, the terminal-oracle π_T^j is considered to be *infected*. WLOG, we limit this query to *compromised* terminals only;
- $\mathtt{SendTerm}(j, M)$, where $j \in \mathbb{N}$ and $M \in \{0,1\}^* \cup \{\mathtt{Start}(\ell)\}$ – This sends message M to π_T^j. A specific $\mathtt{Start}(\ell)$ message asks the terminal to initiate a session, to be done with a user oracle $\pi_{U_\ell}^i$ for a new index i. But only if the terminal-oracle π_T^j is not linked yet to a user, otherwise ℓ is ignored. To compute its response to $\mathcal{A}$, π_T^j may internally talk to its linked human oracle according to the protocol. In addition, if π_T^j is *compromised*, it will additionally give to $\mathcal{A}$ the messages exchanged with its linked human oracle[2].
- $\mathtt{SendServ}(k, M)$, where $k \in \mathbb{N}$ and $M \in \{0,1\}^*$ – This sends message M to oracle π_S^k. The oracle computes the response according to the corresponding algorithm and sends the reply to $\mathcal{A}$.
- $\mathtt{SendHum}(j, M)$ where $j \in \mathbb{N}$ and $M \in \{0,1\}^*$ (and human-readable) – This sends a message to the π_T^j-linked human oracle $\pi_{U_\ell}^i$ on behalf of π_T^j. This is allowed only if the terminal π_T^j is *infected* (and thus *compromised*, which implies the existence of a partenered human oracle). The oracle computes the response according to the corresponding algorithm and sends the reply to $\mathcal{A}$.
- $\mathtt{Test}(j, P)$, where $j \in \mathbb{N}$ and $P \in \{T\} \cup \{S\}$ – If sk_P has been output by π_P^j, then one looks at the internal bit b (flipped once for all at the beginning of the privacy experiment, while $b = 1$ in the authentication experiment). If $b = 1$, then $\mathcal{A}$ gets the real session key sk_P, otherwise it gets a uniformly random session key. This query is only allowed if π_P^j is fresh (defined below).

In the **privacy** experiment, after having adaptively asked several of these oracle queries, the adversary $\mathcal{A}$ outputs a bit b' (a guess on the bit b involved in the $\mathtt{Test}$-queries). The intuition is that the adversary

[2] The messages to the human oracle can be already known to the adversary as they are a function of the oracle's random tape. But we give the adversary the whole communication for convenience.

should not be able to distinguish the real session keys from independent random strings. While in the **authentication** experiment, the goal of the adversary is to make an honest party to successfully complete the protocol execution thinking it "built a secure session" with the right party, whereas that is not the case. In order to formally define the goals and the advantages of the adversary, we present the notions of partnering and freshness, as well as the flags accept and terminate.

FLAGS. In order to model authentication, we follow BPR [BPR00], who defined two flags: accept essentially means that a party has all the material to compute the session key while terminate means that a party thinks that it completes the protocol execution thinking it communicates with the expected other party (a human user in our case). These two flags are initially set to `False`, and they are explicitly set to `True` in the description of the protocol. Note that in Definition 1 U and/or S *accept* if and only if in the end the terminate flag is set to `True`, otherwise, U and/or S *reject*.

PARTNERING. Whereas $\pi^i_{U_\ell}$ and π^j_T are declared as *linked* at the initialization of the communication because of the authenticated channels between users and the terminal, partnering between $\pi^i_{U_\ell}$ and π^k_S is *a posteriori*: they are indeed declared *partners* in the end of the protocol execution if they use the same long-term key and both accept. Then we define partnering between π^j_T and π^k_S, by saying that they are declared *partners* if π^k_S and U^i_ℓ are partners and U^i_ℓ is linked to π^j_T.

FRESHNESS. Informally, the freshness denotes oracles that hold sessions keys that are not trivially known to the adversary.
For $P \in \{T\} \cup \{S\}$, the oracle π^j_P is *fresh*, if no `Test`-query has been asked to π^j_P nor its partner, and none of π^j_P or its partner have been *compromised* (π^j_T is fresh if it has not been compromised, and π^k_S is fresh if the terminal *linked* to the partner human user has not been compromised.)

SECURITY NOTIONS. In the **privacy** security game, the goal of the adversary is to guess the bit b involved in the `Test`-queries. Then we measure the success of an adversary $\mathcal{A}$, that outputs a bit b', by $\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) = 2 \cdot \Pr[b' = b] - 1$. This notion implies *implicit* authentication, which essentially means that no one else than the expected partners share the session key material.
For *explicit* authentication, we define the **authentication** security game: the goal of the adversary is essentially to make a player terminate (flag terminate set to true) without an accepting partner (flag accept set to true). But in our case with *compromised* or even *infected* terminals, this is a bit more complex than usual. We thus split the authentication security in two parts:

– Server-authentication: a user oracle should not successfully terminate a session if there is not exactly one partner server oracle that has accepted. Then, we denote $\mathsf{Adv}^{\mathsf{s\text{-}auth}}_{\mathsf{HAKE}}(\mathcal{A})$ the probability the adversary $\mathcal{A}$ makes such a bad event happens;
– User-authentication: a server oracle should not successfully terminate a session if there is not exactly one partner user oracle that has accepted. Then, we denote $\mathsf{Adv}^{\mathsf{u\text{-}auth}}_{\mathsf{HAKE}}(\mathcal{A})$ the probability the adversary $\mathcal{A}$ makes such a bad event happens.

Eventually, for any adversaries $\mathcal{A}, \mathcal{B}$ there exists an adversary $\mathcal{C}$ against the authentication security for which we define $\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{HAKE}}(\mathcal{C}) = \max\{\mathsf{Adv}^{\mathsf{s\text{-}auth}}_{\mathsf{HAKE}}(\mathcal{A}), \mathsf{Adv}^{\mathsf{u\text{-}auth}}_{\mathsf{HAKE}}(\mathcal{B})\}$.

PASSIVE SESSIONS. We now define a new notion of *passive session*, which extends the `Execute`-queries in the standard BPR model [BPR00]. Recall that `Execute`-queries allow the adversary to get full transcripts of communication between honest parties. Even though the same can be achieved via `Send`-queries, in the security analyses it is useful to count the number of observed honest sessions and the number of maliciously altered sessions separately. In addition, we will not limit to full sessions: the adversary can stop forwarding honest flows, making the session abort. Then, there can be *passive full/partial-sessions*:

Definition 2 (Passive Session). *A (full or partial) session between oracles π^j_T and π^k_S is called* passive*, if the messages of all queries* `SendTerm`$(j, \cdot)$ *or* `SendServ`$(k, \cdot)$ *are either* `Start`$(\cdot)$ *or themselves an output of one of these two queries type. If flows are numbered, this also implies that the actual order*

of flows between T and S has not been modified. If all the outputs have been forwarded as inputs, this is a passive full-session*, otherwise this is a* passive partial-session*.*

Sessions that are not passive are called *active*, since the adversary altered something in the honest execution.

We believe this notion is stronger than the `Execute`-queries defined in the BPR security model, since the adversary does not need to decide from the beginning if all the exchanges will be passive or not. $\mathcal{A}$ can start with a passive sequence and decide at some point to stop (passive partial-session) or behave differently in an adaptive way (active session).

Resources of the Adversary. When doing security analyses, for every adversary and its privacy and authentication advantages, one also has to specify the adversarial resources such as the running time t, the number of oracle queries, the number of player instances, and the numbers $n_{\mathsf{passive}}/n_{\mathsf{active}}$ of (fully) passive and active sessions the adversary needs.

Discussion. We discuss a bit more about our security definitions to explain why they capture the practical threats. First, a passive network adversary is able to observe legitimate communications via `SendServ` and `SendTerm`-queries (these will satisfy the passive sessions definition). An active network adversary can modify legitimate messages or impersonate a terminal or a server by injecting some messages of its choice, again, via `SendServ` and `SendTerm`-queries. This models, in the standard way, possible insecurity (in terms of privacy or authentication) of the network channel between terminals and servers.

Passive-insider attacks (such as keylogger and screen capture malware compromising computers or their browsers) are modeled by `Compromise`-queries followed by `SendTerm`-queries. The former gives the adversary full information about the terminal's internal state, including its random coins and registers' contents, and the latter reveals to the adversary the inputs from the human.

We consider even more powerful attackers who can take full control of the computers or some of their crucial applications such as browsers. In this case, in addition to learning the internal state and all the inputs, the adversary can impersonate the honest terminal while sending adaptively selected messages to the human. We model this by `Infect` and `SendHum`-queries.

Our model captures all the above scenarios and moreover, it takes into account the possibility of multiple simultaneous attacks, such as colluding network and malware adversaries. One can notice that attacks involving `Infect`-queries are stronger than those with `Compromise`-queries: when an adversary infects a terminal, it takes full control on it, with knowledge of its internal state, and thus plays on its behalf, using `SendServ` and `SendHum`-queries.

Note that in any case, we are concerned with the security of a new session, in terms of privacy and authentication, over an honest terminal, that is neither compromised nor infected. Such security should be guaranteed even though the other sessions involving the same human with the same long-term secret were carried over compromised terminals, and if possible even over infected terminals. We model privacy via the `Test`-query and with the appropriate privacy advantage definition. We model authentication via the corresponding advantage definition.

We also stress that we do not consider corruption of the long-term secrets, since they are known by the users and the server only, and we do not allow to corrupt them. Would the long-term secret be leaked, we cannot guarantee any security for future sessions. The interesting open problem of dealing with such corruptions could be addressed using an asymmetric long-term secret: a verifier-based variant that would just provide an encoded version of the user's secret to the server.

3 Building Blocks

For the sake of completeness, the building blocks that will be used in our constructions are detailed in Appendix A. Since most of the details are useful for the proofs only, we just recall or present here the most important descriptions.

3.1 Human-Compatible Function Family

The protocols we propose in the next sections use special function families, which we call *human-compatible (HC)*.

Human-Compatible Function Family: Syntax. A human-compatible (HC) function family is specified by the challenge space $\mathcal{C}$, the key generation algorithm $\mathcal{KG}$, which takes input the security parameter and outputs a key K, and the *challenge-response* function F that takes a key K and a challenge $x \in \mathcal{C}$ and returns the response $r = F_K(x)$. We require that (see Section 2.1 for the definitions):

1. for every K output by $\mathcal{KG}$ and every $x \in \mathcal{C}$, both x and $F_K(x)$ are human-writable and human-readable;
2. $\mathcal{C}$ is human-sampleable.

We also define the Only-Human HC Function Family (where an additional device is excluded), which is the human-compatible function family that also has:

1. for every K output by $\mathcal{KG}$, $F_K(\cdot)$ is human-computable;
2. every K output by $\mathcal{KG}$ is human-memorizable;

Human-Compatible Function Family: Security. In an authentication protocol with challenge-response pairs, intuitively, we would like that any successful authentication to a server should involve an evaluation of the function by the human user. So we expect no compromised/infected terminal to successfully authenticate to the server one more time than it interacted with the human. The security notion from the function is thus a kind of one-more unforgeability [BNPS03]. But here, any query to an $F_K(\cdot)$-oracle should help to immediately answer $F_K(x)$ to the current challenge x, since a second challenge will come from a new session that has closed the previous one, and so the previous challenge is obsolete: the adversary cannot store the $n+1$ challenges, ask n queries, and answer the $n+1$ initial challenges. In our protocols, the adversary gets a random challenge (`GetRandChal`-query), can ask any $F_K(\cdot)$-query (`GetResp`-query), but should answer that challenge (`TestResp`-query), otherwise the failure is detected. After too many failures (recorded in the unvalidated-query counter `ctr`) one may restrict oracle queries. Hence our following security notion which formalizes these restrictions to the adversary.

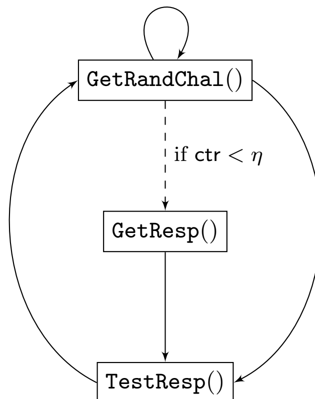

Figure 1. Graph of the sequential oracle calls in the η-unforgeability experiment

η-Unforgeability. As said above, we thus define a kind of sequential one-more unforgeability experiment, with a limit η on the unvalidated-query counter `ctr`, where the queries follow the graph presented on Figure 1. Given an HC function family F, an adversary $\mathcal{A}$, and a public parameter η, one

first generates K with $\mathcal{KG}$ and initializes $\mathtt{ctr} \leftarrow 0$. Then the adversary can ask the following queries, with possible short loops on the `GetRandChal`-query and direct `TestResp`-attempt right after getting the challenge:

1. `GetRandChal`() – It picks a new $x \xleftarrow{\$} \mathcal{C}$, marks it *fresh* and outputs it;
2. `GetResp`(x^*) – If $\mathtt{ctr} < \eta$ and $x^* \in \mathcal{C}$, it returns $F_K(x^*)$ and increments `ctr`. It also marks the *fresh* x as *unfresh*. Otherwise, it outputs $\perp$;
3. `TestResp`(r) –
   - If $F_K(x) = r$ and x is *fresh*, the adversary *wins*;
   - If $F_K(x) = r$ and x is *unfresh*, it decrements `ctr`, marks x as *used*, and outputs 1;
   - Otherwise, it outputs 0.

Because of the sequential iterations, any `TestResp`-query relates to the previous `GetRandChal`-query. One can thus consider one memory-slot to store the challenges, but one only at a time: any new challenge replaces the previous one. The dashed line from `GetRandChal` to `GetResp` emphases the restriction on the number of unvalidated queries. When $\mathtt{ctr} \geq \eta$, the adversary has no more choice than immediately trying an answer for the random challenges. The bound η represents the maximum gap that is allowed at any time between the number of `GetResp`-queries and the number of correct `TestResp`-queries. Note that a random challenge x can only be either *fresh*, *unfresh*, or *used*, and that marking it as one of those erases the other flags. Intuitively, a *fresh* challenge has not been compromised in any way, and succeeding at a `TestResp` on it would indicate the unforgeability has been breached, hence the winning status for the adversary, and the experiment stops. A challenge can switch to the *unfresh* state if the adversary asks the `GetResp`-oracle for an answer. There are only two ways for the experiment to stop: if the adversary wins with a correct `TestResp`-query on a fresh challenge; or if the adversary aborts, it then looses the game. We stress that the adversary can query the `GetResp`-oracle on any x^* of its choice, and so possibly different from the current challenge x obtained with the previous `GetRandChal`-query. But we give it a chance to still answer correctly to the challenge x with the correct `TestResp`-query that, on an *unfresh* challenge, cancels the increment of the counter `ctr`. This counter represents the gap between the number of `GetResp`-queries and the number of correct `TestResp`-queries on random challenges. When one limits `ctr` to be at most 1, any `GetResp`-query should be immediately followed by a correct `TestResp`-query (one-more unforgeability).
This definition is a weaker notion than the one-more unforgeability [BNPS03], but still allows the adversary to exploit malleability: For example, with the RSA function, for a random challenge y, the adversary can ask a `GetResp`-query on any $y' = y \cdot r^e \bmod n$, for an r of its choice, so that it can then extract an e-th root of y. But this would not help it to answer a next fresh challenge.

2-Party η-Unforgeability. Unfortunately, the above clean security notion is not enough for our applications, as client-server situations and man-in-the-middle attacks allow more complex ordering of the queries by the adversary. We therefore present a variant of this experiment below, that is suitable for a protocol involving two parties (hence in the following $b \in \{0, 1\}$).
Given an HC function family F, an adversary $\mathcal{A}$, and a public parameter η, one first generates K with $\mathcal{KG}$ and initializes $\mathtt{ctr} \leftarrow 0$. Then the adversary can ask the following queries:

1. `GetRandChal`(b) – It picks a new $x_b \xleftarrow{\$} \mathcal{C}$, marks it *fresh* and outputs it;
2. `GetResp`(x^*) – If $\mathtt{ctr} < \eta$ and $x^* \in \mathcal{C}$, it returns $F_K(x^*)$ and increments `ctr`. It also marks all *fresh* x_b as *unfresh*. Otherwise, it outputs $\perp$;
3. `TestResp`(r, b) – If x_b exists:
   - If $F_K(x_b) = r$ and x_b is *fresh*, the adversary *wins*;
   - If $F_K(x_b) = r$ and x_b is *unfresh*, it decrements `ctr`, marks x_b as *used* and outputs 1;
   - Otherwise, it outputs 0.

The main difference with the previous experiment are the two memory-slots for challenges x_0 and x_1. But still, any `GetResp`-query must be followed by a correct `TestResp`-query to limit `ctr` from increasing too much.

The advantage of any adversary $\mathcal{A}$ against the unforgeability, $\mathsf{Adv}_F^{\eta\text{-uf}}(\mathcal{A})$ is the probability of winning in the above experiment (with a correct `TestResp`-query on a *fresh* challenge). Such a success indeed means that the adversary found the response for a new random challenge, without having asked for any `GetResp`-query.
The resources of the adversary are the polynomial running time and the numbers $q_\mathsf{c}, q_\mathsf{r}, q_\mathsf{t}$ of queries to `GetRandChal`, `GetResp` and `TestResp` oracles, respectively. Of course it is crucial whether there are secure instantiations of HC function families. We propose some in Section 5.

Indistinguishability. For some constructions, we will expect the sequence of answers $\{F_K(x_i), i = 0, \ldots, T\}$ for challenges x_i (either adversarially chosen or not) to look random, or at least any new element in the sequence is not easy to predict from the previous ones.
For the sake of simplicity, we assume that there exists a global distribution $\mathcal{D}$ with large enough entropy D such that any such sequence is computationally indistinguishable from $\mathcal{D}^{T+1}$: We denote $\mathsf{Adv}_F^{\mathsf{dist}\text{-}c}(\mathcal{D}, \mathcal{A})$ the advantage the adversary $\mathcal{A}$ can get in distinguishing the sequence $\{y_0 = F_K(x_0), \ldots, y_{c-1} = F_K(x_{c-1})\}$ for a random K, from $(y_0, \ldots, y_{c-1}) \xleftarrow{\$} \mathcal{D} \times \ldots \times \mathcal{D}$. For the latter distribution, the probability to guess y_{c-1} from the view of $(y_0, \ldots, y_{c-2})$ is $1/2^D$.
(Weakly) pseudo-random functions definitely satisfy this property. But from a more practical point of view, the function implemented in the RSA SecurID device [RSA] is believed to satisfy it too, with the x_i being a time-based counter.

3.2 Commitment Scheme

We will also use a commitment scheme, a primitive allowing a user to commit on a value x so that the receiver does not learn any information about x, but with the guarantee that the user will not be able to change his mind later.

Commitment Scheme: Syntax and Security. A (non-interactive) commitment scheme $\mathcal{CS}$ is defined by Setup that defines the global public parameters, and two other algorithms:

– $\mathsf{Com}(x)$: on input a message x, and some internal random coins, it outputs a commitment c together with an opening value s;
– $\mathsf{Open}(c, s)$: on input a commitment c and then opening value s, it outputs either the committed value x or $\perp$ in case of invalid opening value.

The **correctness** condition requires that for every x, if $(c, s) = \mathsf{Com}(x)$, then $\mathsf{Open}(c, s)$ outputs x.
The usual **security notions** for commitment schemes are the *hiding* property, which says that x is hidden from c, and the *binding* property, which says that once c has been sent, no adversary can open it in more than one way.
For the security of our protocols we will need additional properties, such as *extractability* (a simulator can extract the value x to which c will be later opened) and *equivocality* (a simulator can generate some fake commitments c it can later open to any x). These features are provided from trapdoors, generated by an alternative setup algorithm and privately given to the simulator. More details can be found in Appendix A. But in the following, we will denote $\mathsf{Adv}_{\mathcal{CS}}(\mathcal{A})$ the advantage an adversary can get against any of these security notions.

Commitment Scheme: Instantiation. An efficient instantiation, with all our expected security properties, in the random oracle model [BR93], can be described as follows:
Given a hash function $\mathcal{H}$ onto $\{0,1\}^\lambda$,

– $\mathsf{Com}(x)$: Generate $r \xleftarrow{\$} \{0,1\}^{2\lambda}$ and output $(c \leftarrow \mathcal{H}(x, r), s \leftarrow (x, r))$;
– $\mathsf{Open}(c, s = (x, r))$: if $\mathcal{H}(s) = c$, return x, otherwise, return $\perp$.

In the random oracle model, this simple scheme is trivially computationally *binding* ($\mathcal{H}$ is collision-resistant) and statistically *hiding* (for a large $r \in \{0,1\}^{2\lambda}$, there are almost the same number of possible r —actually, 2^λ— for any x, that would lead to the commitment c) in the ROM.

3.3 Password-Authenticated Key Exchange

We will also make (black-box) use of a password-authenticated key exchange (PAKE) protocol. A PAKE protocol is an interactive protocol between two parties who share a common low entropy secret (a password) for an execution with session id PAKEsid. At the end of the protocol, the parties output a session key. The correctness requires that any honest PAKE execution with matching passwords results in the parties outputting the same session key.
The security model can be defined in the UC framework [CHK+05], with an ideal functionality $\mathcal{F}_{\mathsf{pake}}$. We will denote $\mathsf{Adv}^{\mathsf{pake}}_{\mathsf{PAKE}}(\mathcal{S},\mathcal{A},\mathcal{Z})$ the advantage the distinguisher $\mathcal{Z}$ can get in distinguishing the ideal world with the simulator $\mathcal{S}$ and the real world with the adversary $\mathcal{A}$. Again, more details can be found in Appendix A, but an efficient instantiation, satisfying all our expected security requirements is the classical EKE [BM92] protocol that encrypts a Diffie-Hellman key exchange, using the password as encryption key. It has been proven UC-secure [ACCP08], under the Computational Diffie-Hellman assumption in the ideal-cipher model.

3.4 Authenticated Encryption

Eventually, for *explicit* authentication of the players, we will make use of an authenticated encryption scheme [BN00] $\mathcal{ES} = (\mathsf{Enc}, \mathsf{Dec})$, where decryption should fail when the ciphertext has not been properly generated under the appropriate key. This will thus provide a kind of key confirmation, as usually done to achieve explicit authentication. However, some critical data will have to be sent, hence a simple MAC would not be enough, privacy of the content is important too.
For an authenticated encryption scheme, there are two main security notions: The *semantic security*, a.k.a `IND-CPA`, prevents any information being leaked about the plaintexts, while the *integrity of ciphertexts*, a.k.a. `INT-CTXT`, essentially says that no valid ciphertext can be produced without the key. The definitions of the corresponding advantages $\mathsf{Adv}^{\mathsf{int\text{-}ctxt}}_{\mathcal{ES}}(\mathcal{A})$ and $\mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathcal{ES}}(\mathcal{B})$, for any adversaries $\mathcal{A},\mathcal{B}$ can be found in [BN00]. In addition, for adversaries $\mathcal{A},\mathcal{B}$ there exists an adversary $\mathcal{C}$ for which we define $\mathsf{Adv}^{\mathsf{authenc}}_{\mathcal{ES}}(\mathcal{C}) = \max\{\mathsf{Adv}^{\mathsf{int\text{-}ctxt}}_{\mathcal{ES}}(\mathcal{A}), \mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathcal{ES}}(\mathcal{B})\}$.
One simple way to achieve secure authenticated encryption is by using a generic Encrypt-then-MAC approach [BN00] or by using a dedicated scheme such as OCB [RBBK01].

4 Generic HAKE Protocols

In this section, we propose two generic HAKE protocols. They build on a simple idea of composing a human-compatible (HC) function family with a password authenticated key exchange (PAKE) protocol. More precisely, a server chooses a random challenge x, the user U_ℓ's response is $r = F_{K_\ell}(x)$, where F is an HC function family and K_ℓ is the long-term secret shared between the user and the server. And finally the terminal and the server execute the PAKE on the one-time password r, as in [PS10]. As already mentioned, whereas the server supports concurrent sessions, since the human does not, there is no sense in maintaining multiple session states for one human user.
However, a straightforward replay attack is possible. The adversary can first just eavesdrop a session by compromising a terminal, and then play on behalf of the server with the observed challenge-response pair (x, r), even when the user uses an honest terminal. The main issue is that there is no reason for the challenge to be distinct in the various sessions if we do not add a mechanism to enforce it. In [PS10]'s constructions, they assume the server is stateful to prevent it. However, we can do better.
This is the goal of our first protocol: it adds a coin-flipping protocol between the terminal and the server to avoid either party to influence the challenge x, and thus to avoid the aforementioned replay attacks. We prove it secure (in terms of privacy, which implies implicit authentication) assuming security of commitments (underlying the coin-flipping), HC function family, and PAKE. However, the concrete security depends on the bound η, which is large enough for our device-based HC function family, but the Only-Human HC function family construction we will propose in Section 5 (and its underlying hardness problem) does not tolerate a high η.

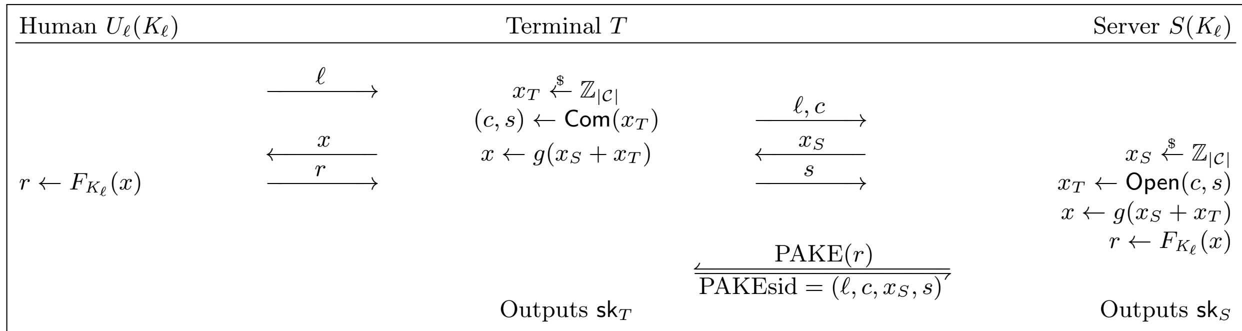

Figure 2. Basic Generic HAKE Construction

Hence, the goal of our second HAKE protocol is to add explicit authentication, which will help limiting the number of malicious challenge-response pairs the adversary can see, or at least to detect them: the user can then suspect the terminal to be infected. We still need the concrete HC function to tolerate at least one malicious challenge, but this remains a reasonable assumption.

4.1 The Basic Generic HAKE

Our first construction makes use of a commitment scheme, an HC function family, and a PAKE.

Description. Let $(\mathcal{KG}, F)$ be a human-compatible function family with challenge space $\mathcal{C}$, let $\mathcal{CS} = (\mathsf{Setup}, \mathsf{Com}, \mathsf{Open})$ be a commitment scheme, let PAKE be a password authenticated key exchange protocol and let $g : \mathbb{Z}_{|\mathcal{C}|} \to \mathcal{C}$ be a bijection. We construct the Basic HAKE $(\mathcal{LKG} = \mathcal{KG}, \mathcal{KE})$. Its interactive $\mathcal{KE}$ protocol is described on Figure 2, here are the descriptions.

- $\mathcal{KE}$ execution:
  1. When the user invokes a terminal to establish a connection with the server, the terminal chooses its part of the challenge x_T, and commits it for the server. It also sends the user's identifier ℓ;
  2. Upon receiving the commitment, the server waits until any previous session for U_ℓ finishes, then it chooses its part of the challenge x_S, and sends it in clear to the terminal;
  3. The terminal then combines both parts x_T and x_S to generate the challenge $x = g(x_S + x_T)$, and asks x to the user;
  4. Upon reading the challenge x, the user computes and writes down the response r for the terminal;
  5. When the terminal receives the response r from the human user, it opens its commitment to the server, and can already starts with the PAKE protocol execution;
  6. Upon receiving the opening value of the commitment, the server opens the latter to get x_T. It can then combine both parts x_T and x_S to generate the challenge $x = g(x_S + x_T)$, and compute the response r. It can then proceed with the PAKE protocol too.

  The terminal and the server both run the PAKE protocol with their (expected) common input r and session id PAKEsid that is the concatenation of the transcript. At the end of the PAKE execution, they come up with two session keys, sk_T and sk_S, respectively, that will be equal if both parties used the same r in the PAKE. Since we do not consider explicit authentication, accept and $\mathsf{terminate}$ flags are not set.

Correctness of the HAKE construction follows from correctness of the building blocks.

Security Analysis. For Basic HAKE, we only assess privacy of the session key, since this protocol does not provide explicit authentication.

Theorem 3. *Consider the Basic HAKE protocol defined in Figure 2. Let $\mathcal{A}$ be an adversary against the privacy security game with static compromises, running within a time bound t and using less than* n_{comp} compromised *terminal sessions,* n_{uncomp} uncompromised *terminal sessions,* n_{serv} *server sessions*

Human $U_\ell(K_\ell)$		Terminal T		Server $S(K_\ell)$
$\mathsf{accept} \leftarrow \texttt{False}$				$\mathsf{accept} \leftarrow \texttt{False}$
$\mathsf{terminate} \leftarrow \texttt{False}$				$\mathsf{terminate} \leftarrow \texttt{False}$
	$\xrightarrow{\ell}$	$x_T \xleftarrow{\$} \mathbb{Z}_{\lvert\mathcal{C}\rvert}$		
		$(c, s) \leftarrow \mathsf{Com}(x_T)$	$\xrightarrow{\ell, c}$	
	$\xleftarrow{x}$	$x \leftarrow g(x_S + x_T)$	$\xleftarrow{x_S}$	$x_S \xleftarrow{\$} \mathbb{Z}_{\lvert\mathcal{C}\rvert}$
$r \leftarrow F_{K_\ell}(x)$	$\xrightarrow{r}$		$\xrightarrow{s}$	$x_T \leftarrow \mathsf{Open}(c, s)$
$\mathsf{accept} \leftarrow \texttt{True}$				$x \leftarrow g(x_S + x_T)$
			$\xleftrightarrow{\text{PAKE}(r)}$	$r \leftarrow F_{K_\ell}(x)$
			$\text{PAKEsid} = (\ell, c, x_S, s)$	
		Parses key: $(k_T \Vert \mathsf{sk}_T)$		Parses key: $(k_S \Vert \mathsf{sk}_S)$
				$\mathsf{accept} \leftarrow \texttt{True}$
$x_U \xleftarrow{\$} \mathcal{C}$	$\xrightarrow{x_U}$	$X_U \leftarrow \mathsf{Enc}_{k_T}(x_U)$	$\xrightarrow{X_U}$	$x_U \leftarrow \mathsf{Dec}_{k_S}(X_U)$
Verifies r_U	$\xleftarrow{r_U}$	$r_U \leftarrow \mathsf{Dec}_{k_T}(R_U)$	$\xleftarrow{R_U}$	$R_U \leftarrow \mathsf{Enc}_{k_S}(F_{K_\ell}(x_U))$
		Outputs sk_T		Outputs sk_S
$\mathsf{terminate} \leftarrow \texttt{True}$	$\xrightarrow{0/1}$			$\mathsf{terminate} \leftarrow \texttt{True}$

Figure 3. Confirmed HAKE Construction

and $n_{\mathsf{active}} \leq n_{\mathsf{comp}} + n_{\mathsf{uncomp}} + n_{\mathsf{serv}}$ active sessions. Then there exist an adversary $\mathcal{B}_1$ attacking the 2-party n_{comp}-unforgeability of the HC function family with $q_{\mathsf{r}}, q_{\mathsf{c}}, q_{\mathsf{t}}$ queries of the corresponding type, an adversary $\mathcal{B}_2$ and a distinguisher $\mathcal{B}_3$ attacking UC-security of the PAKE with a simulator $\mathcal{S}_{\mathsf{pake}}$ as well as an adversary $\mathcal{B}_4$ against the commitment scheme, all running in time t, such that

$$\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq \mathsf{Adv}^{n_{\mathsf{comp}}\text{-}\mathsf{uf}}_{F}(\mathcal{B}_1) + 2 \cdot \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{PAKE}}(\mathcal{S}_{\mathsf{pake}}, \mathcal{B}_2, \mathcal{B}_3) + 6 \cdot \mathsf{Adv}_{\mathcal{CS}}(\mathcal{B}_4) \ ,$$

where $q_{\mathsf{r}} \leq n_{\mathsf{comp}}$, $q_{\mathsf{t}} \leq n_{\mathsf{active}}$, and $q_{\mathsf{c}} \leq n_{\mathsf{uncomp}} + n_{\mathsf{comp}} + n_{\mathsf{serv}}$.

The proof is in Appendix C.

Discussion. Concrete security of the HC function family is definitely the most crucial compared to that of other building blocks, since it is hard to balance strong security and usability. This is why we emphasize this in the above theorem.

We note that the sessions which lead to `TestResp`-queries have *non-oracle-generated* flows and therefore correspond to classical on-line dictionary attacks: the adversary simply tries to impersonate the user/terminal to the server (or vice-versa), with a guess for the answer r (unless the query has been asked to the `GetResp`-oracle). Indeed, sessions with `GetResp`-queries on the exact challenges have *compromised* terminals and correspond to a spyware keylogger that records random challenge-response pairs. If using such a compromised terminal can be considered rare, this remains reasonable. Eventually, the sessions which lead to `GetResp`-queries on different challenges are the most critical, but they should likely fail. And we expect them to be quite exceptional. As remarked above, such sessions likely conclude to a failure: no concrete session is established with the server the user wanted to connect to (if the HC function family is still secure after such adaptive queries). If the user can detect such a failure, he can run away from this terminal. We now propose a way for the user to detect such a dangerous terminal, and thereafter take appropriate measures. At the same time, our next proposal will achieve *explicit* authentication.

4.2 The Confirmed HAKE

We now enhance the Basic Generic HAKE by adding two confirmation flows (see Figure 3) that allow the user to detect a bad behavior of the adversary, who compromised the device, and take appropriate measures. This can happen in two different scenarios: the adversary just compromised a terminal and additionally plays on behalf of the server, which allows it to ask any query to the user through the terminal, or the adversary infected a terminal that allows it to directly ask any query to the user.

As said above, such dangerous cases lead to no connection with the expected server. The user will thus check whether he built a secure session with the expected server, who should be able to answer

to a fresh random challenge. This is performed under the fresh key, established with the PAKE, using a secure authenticated encryption. As shown below, the two additional flows will not only provide *explicit* authentication, but also allow the user to detect such bad events and take measures. For this, it is important that the user does not start multiple sessions concurrently, which is anyway not realistic for a human (as already noticed above).

Description. The protocol is similar to Basic HAKE, but it uses an additional building block, an authenticated encryption scheme $\mathcal{ES} = (\mathsf{Enc}, \mathsf{Dec})$, that is used in the new last stage of the protocol. The description is in Figure 3.
Since we now consider the authentication of the players, we additionally include accept and terminate flags in the protocol: The user U_ℓ accepts after sending the first response while the server S accepts after the PAKE. Then both terminate when they have the confirmation of the other partner. More precisely, the user terminates after sending the last bit (1 for acceptance and 0 for rejection) to the terminal (thus having verified the server's response in the last stage), and the server terminates after sending the encrypted response (thus having checked the terminal can generate a valid ciphertext).
Note that if the protocol terminates, sk_T and sk_S must be equal, since our additional flows act as confirmation flows for the PAKE.

Security Analysis. We now present Theorem 4 regarding the security of our Confirmed HAKE in the HAKE privacy and authenticity experiment.
While it relies on the same security properties of PAKE, authenticated encryption, commitment scheme and HC function, a critical parameter is added, the number of human sessions that *reject* in the end. Indeed, the explicit authentication property we achieve means that any attempt at issuing an adaptive query unrelated to the challenge will likely lead to a failure of the PAKE protocol, that can in turn be detected by the human, as he doesn't get the answer to x_U he looked for. This allows to use a much stricted η in the HC unforgeability game (even $\eta = 1$ for a very strict human user), which is a much more reasonable goal for an HC function family such as the one derived from [BBDV16], that we will present in Section 5.2

Theorem 4. *Consider the Confirmed HAKE protocol defined in Figure 3. Let $\mathcal{A}, \mathcal{A}'$ be adversaries against the privacy and authenticity security game of HAKE within a time bound t and using less than n_{comp} compromised terminal sessions, n_{uncomp} uncompromised terminal sessions, n_{serv} server sessions, $n_{\mathsf{active}} \leq n_{\mathsf{comp}} + n_{\mathsf{uncomp}} + n_{\mathsf{serv}}$ active sessions and n_{hr} human session that reject in the end. Then there exist two adversaries $\mathcal{B}_1, \mathcal{B}'_1$ attacking the 2-party $(n_{\mathsf{hr}}+1)$-unforgeability of HC function family with $q_{\mathsf{r}}, q_{\mathsf{c}}, q_{\mathsf{t}}$ queries of the corresponding type, two adversaries $\mathcal{B}_2, \mathcal{B}'_2$ and two distinguishers $\mathcal{B}_3, \mathcal{B}'_3$ attacking UC-security of the PAKE with the simulator $\mathcal{S}_{\mathsf{pake}}$, two adversaries $\mathcal{B}_4, \mathcal{B}'_4$ against the authenticated encryption, as well as two adversaries $\mathcal{B}_5, \mathcal{B}'_5$ against the commitment scheme, all running in time t, such that*

$$\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq \mathsf{Adv}^{(n_{\mathsf{hr}}+1)\text{-}\mathsf{uf}}_{F}(\mathcal{B}_1) + 2 \cdot \mathsf{Adv}^{\mathsf{authenc}}_{\mathcal{ES}}(\mathcal{B}_4) + 6 \cdot \mathsf{Adv}_{\mathcal{CS}}(\mathcal{B}_5) + 2 \cdot \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{PAKE}}(\mathcal{S}_{\mathsf{pake}}, \mathcal{B}_2, \mathcal{B}_3),$$
$$\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{HAKE}}(\mathcal{A}') \leq \mathsf{Adv}^{(n_{\mathsf{hr}}+1)\text{-}\mathsf{uf}}_{F}(\mathcal{B}'_1) + 2 \cdot \mathsf{Adv}^{\mathsf{authenc}}_{\mathcal{ES}}(\mathcal{B}'_4) + 6 \cdot \mathsf{Adv}_{\mathcal{CS}}(\mathcal{B}'_5) + 2 \cdot \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{PAKE}}(\mathcal{S}_{\mathsf{pake}}, \mathcal{B}'_2, \mathcal{B}'_3),$$

where $q_{\mathsf{r}} \leq 2n_{\mathsf{comp}}$, $q_{\mathsf{t}} \leq n_{\mathsf{active}}$, $q_{\mathsf{c}} \leq n_{\mathsf{comp}} + n_{\mathsf{uncomp}} + n_{\mathsf{serv}}$.

The proof is provided in Appendix D.

Remark 5. We also note that given the confirmation phase, and assuming the strong policy of resetting all credentials if the confirmation phase fails, the coin-flipping part is no longer necessary for the security proof: we could let the server choose the challenge during the first phase and the human in the second one (to avoid one player being to make replay attacks). We chose to keep it as part of the protocol because, first, this would not reduce the number of flows since the terminal always initiates such a connection, and second, without coin-flipping a network attacker could test adaptive challenges. The confirmation phase would fail, but there is no real need for the user to take severe measures and change the long-term secret in such a weak attack. Hence we prevent adaptive tests (from network attacks)

with coin-flipping, which may be useful if a policy a little weaker is in use, such as resetting only if there is suspicion of terminal infection.

5 Human-Compatible Function Family Instantiation

In Section 3, we proposed instantiations for the HAKE building blocks, except for the HC function family. We now focus on the latter in this section.

5.1 Token-Based HC Function Family Instantiation

First, we introduce a simple token-based HC function family. This assumes that the human is in possession of a simple device on which it can input challenge x and get the response $r \leftarrow F_K(x)$. The device will store K and perform the computation, but the human is still responsible for the communication with the terminal.
This allows us to use strong cryptographic primitives. For instance, we could set $K \xleftarrow{\$} \{0,1\}^\lambda$ and $F_K : [\![0,9]\!]^{t'} \to [\![0,9]\!]^t$ a pseudorandom function. In the random oracle model (for modeling $\mathcal{H}$ in $F_K(x) = \mathcal{H}(K \| x)$), we have $\mathsf{Adv}_F^{\eta\text{-uf}}(\mathcal{A}) \leq 10^{-t}$ for any adversary and any η, since an adversary can just guess by chance the answer to a fresh challenge. Note that this function is obviously *human-readable*, *human-writable* and *human-sampleable* as its input/output are numbers in basis 10 so it is an HC function family.
Hence this function family is a good candidate to use in our Basic generic HAKE protocol from Section 4.1 or its simplified version from Section 6.1.

5.2 Only-Human HC Function Family Instantiation

However, avoiding such devices would be much better in practice. We are thus interested in the Only-Human HC function family that would not require anything beyond simple human memory and brain computation power. Since such a function is necessarily weaker, we will use it in our confirmed HAKE protocol from Section 4.2, that has a much tighter control over adaptive queries and therefore requires weaker security properties from the HC function family.

Construction We present a candidate based on the construction of Blocki et al [BBDV16], which security is based on [FPV15a]: Consider a challenge space $\mathcal{C} = \mathcal{X}_l^t \subseteq [n]^{lt}$, where $[n] = \{1, \ldots, n\}$ is the set of n integers each representing one of the n variables and $\mathcal{X}_l$ denotes the space of ordered clauses of l variables without repetition. The parameter t indicates that each challenge consists of t independent clauses, i.e., "small" challenges. The key generation algorithm $\mathcal{KG}$ of our HC function family takes as input a parameter n, then outputs a random mapping $\sigma : [n] \to \mathbb{Z}_d$ as the key K, where the integer d is a constant. Usually we set $d = 10$ because most humans are familiar with computations on digits. Let $\sigma^l : [n]^l \to \mathbb{Z}_d^l = (\sigma, \cdots, \sigma)$ denote the mapping that applies σ to each element of an l-tuple. Using a public human-computable function $f : \mathbb{Z}_d^l \to \mathbb{Z}_d$ that is instantiated later, the challenge-response function F takes a key $K = \sigma$ and a challenge $x \in \mathcal{C}$ as inputs, and returns a response $r = F_K(x)$. Here $F_K : \mathcal{C} \to \mathbb{Z}_d^t$ is defined as a t-tuple $(t \geq 1)$ $(f \circ \sigma^l, \cdots, f \circ \sigma^l)$, where $\circ$ indicates the function composition.
For instance, if $n = 100$, $l = 3$, $d = 10$, $t = 2$, $x = ((1,4,20),(3,36,41))$, $\sigma(i) = (i+3) \mod 10$ and $f = (x_1 - x_2 + x_3) \mod 10$, then $\sigma((1,4,20)) = (4,7,3)$, $\sigma((3,36,41)) = (6,9,4)$ and $F_K(x) = (0,1)$.
Given integers $k_1, k_2 > 0$, the function f is instantiated as $f_{k_1,k_2} : \mathbb{Z}_{10}^{10+k_1+k_2} \to \mathbb{Z}_{10}$, which is defined as follows:

$$f_{k_1,k_2}(x_0, \ldots, x_{9+k_1+k_2}) = x_{\left(\sum_{i=10}^{9+k_1} x_i \mod 10\right)} + \sum_{i=10+k_1}^{9+k_1+k_2} x_i \mod 10.$$

Note that when f is instantiated as f_{k_1,k_2}, we have $l = 10 + k_1 + k_2$ and $d = 10$.

It is easy to see that such a function family is an Only-Human HC function family apart from the human memorization property. However, we can allow for images to represent the variables. As illustrated in [BBDV16], by using mnemonic helpers, humans are able to remember such mappings from images to digits. As an evidence, the primary author of [BBDV16] was able to memorize a mapping from $n = 100$ images to digits in 2 hours.

Security In [BBDV16], the authors proved the intractability to answer to a new random challenge for the above HC function family instantiation based on the conjecture about the hardness of *random planted constraint satisfiability problems (RP-CSP)*. We briefly recall a special case of the RP-CSP conjecture, which we call the RP-CSP* conjecture, and its implied security theorem, both with our notations. For an in-depth review of those notions, the reader should refer to [BBDV16].

The RP-CSP* Conjecture. Before stating this conjecture, we introduce some notations as in [BBDV16]. Denote $H(\alpha_1, \alpha_2) = |\{i \in [n] \mid \alpha_1[i] \neq \alpha_2[i]\}|$ as the Hamming distance between two strings $\alpha_1, \alpha_2 \in \mathbb{Z}_d^n$. Use $H(\alpha) = H(\alpha, \mathbf{0})$ to denote the Hamming weight of α. Then we say two mappings $\sigma_1, \sigma_2 \in \mathbb{Z}_d^n$ are *ε-correlated* if $H(\sigma_1, \sigma_2)/n \leq (d-1)/d - \varepsilon$.

Conjecture 1 (RP-CSP*) *Consider the function f_{k_1,k_2} described above, for any $\varepsilon, \varepsilon' > 0$ and any probabilistic polynomial time (in n) adversary $\mathcal{A}$, there exists an interger $N \in \mathbb{N}$, such that for all $n > N$, $m \leq n^{\min\{(k_2+1)/2, k_1+1-\varepsilon'\}}$, we have $\mathsf{Adv}^{\mathsf{rand}}_{f_{k_1,k_2}}(\mathcal{A}, \varepsilon) = \mathsf{negl}(n)$, where $\mathsf{Adv}^{\mathsf{rand}}_{f_{k_1,k_2}}(\mathcal{A}, \varepsilon)$ is the probability that $\mathcal{A}$ outputs a mapping σ' that is ε-correlated with the secret mapping σ given m random "small" challenge-response pairs $\{(C_i, f_{k_1,k_2}(\sigma^l(C_i)))\}_{1 \leq i \leq m}$.*

Remark 6. The RP-CSP conjecture in [BBDV16] is a general version of the RP-CSP* Conjecture 1, where f can be instantiated as other functions. Here, for simplicity, we only state the conjecture where $f = f_{k_1,k_2}$. In [BBDV16], the authors also prove strong evidence in support of the RP-CSP conjecture: it holds for any statistical adversary and any Gaussian Elimination adversary. As observed in [FPV15b], most natural algorithmic techniques have statistical analogues except the Gaussian Elimination.

Basic η-Unforgeability. To state the security theorem in [BBDV16], we need the following "basic" HC security notion that is a "non-malleable" version of the η-unforgeability. It indeed assumes that asking a `GetResp`-query with an input different from the current random challenge should not help to answer this challenge correctly to the `TestResp`-query. Given an HC function family F, an adversary $\mathcal{A}$, and a public parameter η, one first generates K with $\mathcal{KG}$ and initializes $\mathsf{ctr} \leftarrow 0$. Then the adversary can ask the following queries:

1. `GetRandChal`() – It picks a new $x \xleftarrow{\$} \mathcal{C}$, marks it *fresh* and outputs it;
2. `GetResp`(x^*) – It increments ctr if $x^* \neq x$;
   - If $\mathsf{ctr} \leq \eta$ and $x^* \in \mathcal{C}$, it outputs $F_K(x^*)$ and marks x as *unfresh*;
   - Otherwise, it outputs $\perp$;
3. `TestResp`(r) –
   - If $F_K(x) = r$ and x is *fresh*, the adversary *wins*;
   - If $F_K(x) = r$ and x is *unfresh*, it outputs 1;
   - Otherwise, it outputs 0.

Just like the η-unforgeability experiment, the above oracle calls are sequential (similar to Figure 1), starting with a `GetRandChal`-query. But since non-malleability is assumed, only `GetResp`-queries with inputs different from the current random challenges make the counter increase, and it is never decreased. The advantage of any adversary $\mathcal{A}$ against the above unforgeability, $\mathsf{Adv}_F^{\eta\text{-uf-basic}}(\mathcal{A})$ is the probability of winning in the above experiment. Such a success indeed means that the adversary found the response for a new random challenge, without having asked for a `GetResp`-query. The parameter η restricts the number of "adaptive" `GetResp` queries that $\mathcal{A}$ can make, where adaptive means "different from the current random challenge".

The resources of the adversary are the polynomial running time and the numbers $q'_{\mathsf{c}}, q'_{\mathsf{r}}, q'_{\mathsf{t}}$ of queries to the above `GetRandChal`, `GetResp` and `TestResp` oracles, respectively. For convenience, denote by q''_{t} the number of `TestResp`-queries such that the current random challenge x is *fresh*. By definition, we have $q'_{\mathsf{r}} \leq q'_{\mathsf{c}}$, $q'_{\mathsf{t}} \leq q'_{\mathsf{c}}$ and $q''_{\mathsf{t}} \leq q'_{\mathsf{c}} - q'_{\mathsf{r}}$.

HC Function Family Security Results. Under Conjecture 1, one can prove the following unforgeability result about the HC function family.

Theorem 7 (From [BBDV16]). *Given $\varepsilon, \varepsilon' > 0, t \in \mathbb{N}_+$ and $\delta > (\frac{1}{10} + \varepsilon)^t$, for any probabilistic polynomial time (in $n, q'_{\mathsf{c}}, 1/\varepsilon$) adversary $\mathcal{A}$ against the basic 0-unforgeability security of the HC function family F constructed above using $f = f_{k_1,k_2}$ with*

$$q''_{\mathsf{t}} = 1, q'_{\mathsf{c}} \leq \frac{1}{t} \cdot n^{\min\{(k_2+1)/2, k_1+1-\varepsilon'\}} - 1,$$

under Conjecture 1, we have $\mathsf{Adv}_F^{\text{0-uf-basic}}(\mathcal{A}) < \delta$.

Note that in the basic 0-unforgeability security game, the adversary learns nothing from $\mathtt{GetResp}(x^*)$ if x^* is not the current random challenge x. So if $\eta = 0$, the adversary $\mathcal{A}$ is only given random challenge-response pairs.
This result is actually not strictly good-enough, even for our confirmed HAKE. Indeed, if the function does not allow for at least one adaptive query, an attacker could make it in the first exchange (using an infected terminal), then break the unforgeability of the function before the confirmation flow and make the protocol succeed, hence avoiding detection. Thus, we extend the RP-CSP conjecture to allow $\log n$ adaptive "small" challenge-response pairs.

Conjecture 2 *Consider the function f_{k_1,k_2} described above, for any $\varepsilon, \varepsilon' > 0, t \in \mathbb{N}_+$ and any probabilistic polynomial time (in n) adversary $\mathcal{A}$, there exists an interger $N \in \mathbb{N}$, such that for all $n > N$, $m_r \leq n^{\min\{(k_2+1)/2, k_1+1-\varepsilon'\}}$ and $m_a \leq t \log n$, we have $\mathsf{Adv}^{\mathsf{adapt}}_{f_{k_1,k_2}}(\mathcal{A}, \varepsilon) = \mathsf{negl}(n)$, where $\mathsf{Adv}^{\mathsf{adapt}}_{f_{k_1,k_2}}(\mathcal{A}, \varepsilon)$ is the probability that $\mathcal{A}$ outputs a mapping σ' that is ε-correlated with the secret mapping σ given m_r random "small" challenge-response pairs and the correct responses to m_a "small" challenges adaptively chosen by $\mathcal{A}$.*

Proof. For any adversary $\mathcal{A}$ we can construct an adversary $\mathcal{B}$ such that $\mathsf{Adv}^{\mathsf{adapt}}_{f_{k_1,k_2}}(\mathcal{A}, \varepsilon) \leq 10^{t \log n} \times \mathsf{Adv}^{\mathsf{rand}}_{f_{k_1,k_2}}(\mathcal{B}, \varepsilon)$.
$\mathcal{B}$ simulates $\mathcal{A}$'s view by providing $\mathcal{A}$ with the given m_r random "small" challenge-response pairs and randomly guessing the responses to the m_a ($\leq t \log n$) adaptive "small" challenges. The probability of correctly guessing all adaptive ones is $10^{-t \log n}$ (refer to the construction of f_{k_1,k_2}), hence the above advantage reduction. One should note that $10^{t \log n} \times \mathsf{negl}(n) = \mathsf{negl}(n)$ and $\mathcal{B}$'s running time is polynomial in n.

Under this extended conjecture, one can prove the following stronger unforgeability result about the HC function family, which "almost" suits our confirmed HAKE (see Figure 3):

Theorem 8. *Given $\varepsilon, \varepsilon' > 0, t \in \mathbb{N}_+$ and $\delta > (\frac{1}{10} + \varepsilon)^t$, for any probabilistic polynomial time (in $n, q'_{\mathsf{c}}, 1/\varepsilon$) adversary $\mathcal{A}$ against the basic η-unforgeability security of the HC function family F constructed above using $f = f_{k_1,k_2}$ with*

$$\eta \leq \log n, q'_{\mathsf{c}} \leq \frac{1}{t} \cdot n^{\min\{(k_2+1)/2, k_1+1-\varepsilon'\}} - 1,$$

under Conjecture 2, we have $\mathsf{Adv}_F^{\eta\text{-uf-basic}}(\mathcal{A}) < q''_{\mathsf{t}} \cdot \delta$.

Proof. The proof is almost the same as that of Theorem 7. Informally, we need to show that any adversary $\mathcal{A}$ that breaks the basic η-unforgeability security of the HC function family can also "recover"

the secret mapping σ in Conjecture 2. The reader can refer to the proof of Theorem 5 in [BBDV16], which we call the "HCP" proof below, for the details.
Nevertheless, here the theorem differs from Theorem 7 in several aspects. First, $\mathcal{A}$ can adaptively select $t \log n$ "small" challenges to get the correct responses, while in Theorem 7 only random ones are allowed. But having adaptive queries does not affect the HCP proof because it only uses $\mathcal{A}$ as a blackbox to predict the responses to any t "small" challenges. Second, we apply an union bound of q''_{t} queries to the final advantage.

Remark 9. In the above theorem $\varepsilon, \varepsilon'$ are almost 0. We can set $n = 100$, $k_1 = 1$, $k_2 = 3$ and $t = 5$, then $\eta \leq 6$, $q'_{\mathsf{c}} \leq n^2/t - 1 \approx 2000$ and $\mathsf{Adv}_F^{\eta\text{-uf-basic}}(\mathcal{A}) < q''_{\mathsf{t}} \cdot 10^{-t} \leq 1/50$.

We believe a similar theorem holds for $\mathsf{Adv}_F^{\eta\text{-uf}}$ (by replacing the oracles with those in the 2-party η-unforgeability experiment), which our HAKE security can rely on. The intuition is as follows. With the HC function family instantiation described in this section, a $\mathtt{GetResp}(x^*)$ query in the η-unforgeability experiment should not have x^* too "far" from the random challenge x output by the latest $\mathtt{GetRandChal}$ query. Otherwise, it is very unlikely for the adversary to guess correctly in the $\mathtt{TestResp}$ query. But the adversary can modify x a little bit to guess the correct response with a smaller failure probability. This is the difference between the two unforgeability notions: the basic one does not tolerate any malleability, whereas the other can exploit malleability. Because of the size of the challenge space, that has to be quite large (it is essentially $n^{t(10+k_1+k_2)}$, and thus 2^{465}, with the above parameters), the number of challenges that are "close" to any random challenge accounts for a tiny proportion. Thus, the adversary should not get much help from such "nearly random" challenges. Besides, such queries risk increasing the counter in the $\mathtt{GetResp}$ oracle without extracting much useful information. In addition, the two memory slots will not increase much the advantage of an adversary, and so $\mathsf{Adv}_F^{\eta\text{-uf-basic}}$ and $\mathsf{Adv}_F^{\eta\text{-uf}}$ should be quite close for this specific HC function family instantiation. We leave further studies of security of the HC function family from [BBDV16] to future works.

6 Device-Assisted HAKE Protocols

In this section, we take a step back from Only-Human HC function family to allow the use of an additional device that will perform the computations in place of the human. In this setting, the HC function family can be quite powerful and thus resist to many adaptive queries. We consider it in two scenarios: first in a similar context as the Basic Generic HAKE, where one can enter a challenge onto the device to get the response; and second, a time-based token, that outputs the response every timeframe, with the time as the challenge (without having the user to enter it).

6.1 Simplified Basic HAKE

According to the security proof of the Basic Generic HAKE, the PAKE has to be instantiated with a UC-Secure protocol, which turns out to be quite costly. Indeed, the only efficient scheme that achieves this security level is the encrypted key exchange protocol (EKE) [BM92]. However, the proof holds in the Ideal-Cipher model, for a symmetric blockcipher that should only output elements in the Diffie-Hellman group. In practice, the best way to do it is to iterate a large blockcipher until one falls in the group. First, a large blockcipher from a hash function (modeled as a random oracle) has fueled a whole line of works [CPS08, HKT11, DS16], and is nevertheless already quite costly: at the time of writing, at least 8-round Feistel network is required [DS16], with an impossibility result below 6 [CPS08]. Thereafter, additional iterations are required to build a permutation onto the group. This thus eventually corresponds to dozens of hash function evaluations.
Looking back at the construction, using a full PAKE seems anyway as a bit of an overkill since the ephemeral secrets are only used once, and need not to be kept secret afterwards. We present in Appendix B a protocol that uses commitments instead of a full PAKE to achieve better efficiency.

Time	Human U_ℓ		Terminal T		Server S
$\leq t$	$\mathsf{accept} \leftarrow \mathtt{False}$				$\mathsf{accept} \leftarrow \mathtt{False}$
					$\mathsf{terminate} \leftarrow \mathtt{False}$
t		$\xrightarrow{\ell}$	$x_T \xleftarrow{\$} \mathbb{Z}_p, X_T \leftarrow g^{x_T}$		$x_S \xleftarrow{\$} \mathbb{Z}_p, X_S \leftarrow g^{x_S}$
t		$\xrightarrow{\mathsf{pw}_t}$	$(c_T, s_T) \leftarrow \mathsf{Com}_T(X_T, \mathsf{pw}_t)$	$\xrightarrow{\ell, X_T, c_T}$	
t	$\mathsf{accept} \leftarrow \mathtt{True}$			$\xleftarrow{X_S, c_S}$	$(c_S, s_S) \leftarrow \mathsf{Com}_S(X_S, \mathsf{pw}_t)$
$\vdots$			Wait for timeframe $> t$		Wait for timeframe $> t$
$> t$				$\xrightarrow{s_T}$	If $\mathsf{Open}_T(c_T, s_T) = (X_T, \mathsf{pw}_t)$ and $(\ell, t) \notin \Lambda$, store (ℓ, t) in Λ
$> t$					Otherwise reject
$> t$					$\mathsf{accept} \leftarrow \mathtt{True}$
$> t$			Reject if $\mathsf{Open}_S(c_S, s_S) \neq (X_S, \mathsf{pw}_t)$	$\xleftarrow{s_S}$	Outputs $(X_T)^{x_S}$
$> t$			Outputs $(X_S)^{x_T}$		$\mathsf{terminate} \leftarrow \mathtt{True}$

Figure 4. Time-Based Device-Assisted HAKE Construction

6.2 Time-Based HAKE

Scenario. In this section, we focus on the particular (but quite usual) case where the physical device does not have a dedicated input but uses time instead to compute its output. More precisely, our protocol considers a device, such as the RSA-SecurId [RSA] token, that, based on an internal seed (the long-term key K_ℓ), generates a one-time password (the value $F_{K_\ell}(t)$, based on the time period t), and displays it on an LCD-Screen. The password is tied to an internal clock, and changes every τ (e.g. 30s). Note that such a password is already *human readable* and *human writable*, hence it satisfies our human-compatible communications.

Building on the security model presented in Section 2.2, we now consider time as a variable, that is to be segmented into timeframes (each spanning τ seconds). We then number those timeframes and associate to each message sent between T and S this number, representing the fact that each party can measure time and identify the timeframe in which the message was sent.

Since the one-time passwords are generated by a secure device implementing F_{K_ℓ}, we can make the assumption that, for each timeframe, the output is indistinguishable from an element sampled from the distribution $\mathcal{D}$ with entropy greater than D (which increases the advantage of an adversary $\mathcal{A}$ by at most $\mathsf{Adv}_F^{\mathsf{dist}\text{-}T}(\mathcal{A})$ after T timeframes).

We rely on the requirement that any user U_ℓ can only make use of one terminal during a timeframe. That is, he may not attempt to authenticate using more than one terminal in a single time period.

Protocol. We now propose a protocol for time-based device-assisted HAKE. It is presented on Figure 4. As in the previous Simplified Basic HAKE, it makes use of a commitment scheme on top of the unauthenticated Diffie-Hellman scheme to perform authentication.

The commitment scheme $\mathcal{CS}$ is initialized twice, with two independent setup, leading to $\mathsf{Com}_T/\mathsf{Open}_T$ and $\mathsf{Com}_S/\mathsf{Open}_S$, each of them being used for the commitments generated by the terminal and the server, respectively. We also setup a group $\mathbb{G}$ of prime order p in which the discrete logarithm problem is believed to be hard. Let g be a generator of $\mathbb{G}$.

The protocol itself is split into two parts: the *commitment* phase which must happen during a timeframe t (that we will call the *session timeframe*) and the *verification* phase, that must happen later than the session timeframe.

This delay is a clear limitation on the total speed of the protocol, which on average will take $\tau/2$. It will however prove necessary, as it allows $F_{K_\ell}(t)$ to be revealed without compromising the security of the scheme, therefore building on the one-time specificity of the password. To enforce a unique session in a timeframe, the server will not accept to run several sessions within the same timeframe, with the same user, as the latter should not do it anyway (see above). This would thus come from an adversary, and then allowing multiple sessions in a timeframe t can compromise other sessions in the same timeframe when $\mathcal{F}_{K_\ell}(t)$ is revealed.

Scheme	Flows	Terminal		Server		Communication
		expon.	$\mathcal{H}$ eval.	expon.	$\mathcal{H}$ eval.	complexity
1(SPAKE1) [PS10, AP05]	4	3	1	3	1	4λ
This work	4	2	2	2	2	10λ

Table 1. Performance of the Time-Based Device-Assisted HAKE

It is interesting to note that this protocol uses the time period t as the HAKE challenge (the challenge is a counter) and the one-time password ($F_{K_\ell}(t)$) read from the device as the human's response. Therefore, partnering between U and S is entirely determined at the end of the session timeframe t.

Security Analysis. In the security analysis, as in Section 4, we only consider static *compromises*. Hence $\texttt{Compromise}(j)$ can only be the first oracle query of a session, and $\texttt{Infect}(j)$ can only affect *compromised* sessions. Since *compromises* are known before the first flow and partnering between Human and Server is determined at the end of timeframe t, this means that *freshness* itself can be perfectly ascertained in any timeframe $> t$.
The security of our protocol heavily relies on the strong-security of the commitment scheme (see Section 3 and Appendix A).

Theorem 10. *Consider the Time-Based Device-Assisted HAKE protocol defined in Figure 4. Let $\mathcal{A}, \mathcal{A}'$ be an adversaries against the privacy and user authenticity security games with static compromises, running within time t and using less than n_{serv} non-passive sessions against the server oracle, n_{term} non-passive sessions against the terminal oracle, $n_{\mathsf{total}} > n_{\mathsf{term}} + n_{\mathsf{serv}}$ total sessions and $T < n_{\mathsf{total}}$ unique timeframes. Then there exist an adversary $\mathcal{B}_1$ against the indistinguishability of the password-distribution $\mathcal{D}$ running in time t, an adversary $\mathcal{B}_3$ against the DDH experiment running in time $t + 8n_{\mathsf{total}}\tau_{exp}$, and adversary $\mathcal{B}_2$ against the commitment scheme running in time t:*

$$\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \cdot 2^{-D} + \mathsf{Adv}^{\mathsf{dist}\text{-}T}_{F}(\mathcal{B}_1) + \mathsf{Adv}^{\mathsf{ind}}_{DDH}(\mathcal{B}_3) + (n_{\mathsf{total}} + 3) \cdot \mathsf{Adv}_{\mathcal{CS}}(\mathcal{B}_2),$$
$$\mathsf{Adv}^{\mathsf{u\text{-}auth}}_{\mathsf{HAKE}}(\mathcal{A}') \leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \cdot 2^{-D} + \mathsf{Adv}^{\mathsf{dist}\text{-}T}_{F}(\mathcal{B}_1) + (n_{\mathsf{total}} + 2) \cdot \mathsf{Adv}_{\mathcal{CS}}(\mathcal{B}_2),$$

with τ_{exp} the time necessary to exponentiate one group element, and n_{total} the global number of sessions.

The proof is in Appendix E.

Remark 11. Note that the Time-Based Device-Assisted HAKE only achieves user authentication in our setting, since *server authentication* requires the server identity to be approved by the human in our setting (the terminal could be *infected* so it cannot be relied on). A similar approach to the one of the Confirmed HAKE could be used to achieve a full mutual authentication.

Perfomances. We offer in Table 1 a comparison (in terms of numbers of flows, exponentiations, $\mathcal{H}$ evaluations and overall communication complexity) of the performances of our HAKE protocol with the one-time PAKE 1(P) construction of [PS10], instantiated with SPAKE1 from [AP05] as a reference. Since SPAKE1 is also proven in the random oracle model, it is fair to use the efficient commitment scheme described in Section 3. We do not include the redundant X_P in s_P (it is transmitted at the commitment stage) for the communication complexity, and for a security parameter λ, we assume the group elements to be encoded into 2λ-long bit-strings.
While our communication complexity is higher, the computational load is reduced by 30% from [PS10] with the most efficient PAKE. Relaxing the PAKE security properties allows a significant gain from the complexity point of view.

7 Conclusion

We proposed the first user authenticated key exchange protocols which can tolerate corrupted terminals: if a user happens to log in to a server from a terminal that has been fully compromised, then the other

past and future user's sessions initiated from honest terminals stay secure. We formalized security for Human Authenticated Key Exchange (HAKE) protocols and proposed generic constructions based on human-compatible (HC) function families or small auxiliary devices such as RSA SecurID, password-authenticated key exchange (PAKE), commitment, and authenticated encryption. We analyzed security of our HAKE protocols and discussed their instantiations. We left several interesting open problems and believe that our work will promote further developments in the area of human-oriented cryptography.

Acknowledgments

We thank the reviewers for insightful comments. Alexandra Boldyreva and Shan Chen are supported in part by the National Science Foundation under Grants No. CNS-1318511 and CNS-1422794. Pierre-Alain Dupont and David Pointcheval are supported in part by the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013 Grant Agreement no. 339563 – CryptoCloud).

References

ABB+13. M. Abdalla, F. Benhamouda, O. Blazy, C. Chevalier, and D. Pointcheval. SPHF-friendly non-interactive commitments. In *ASIACRYPT 2013, Part I*, *LNCS* 8269, pages 214–234. Springer, Heidelberg, December 2013. (Page 26.)

ACCP08. M. Abdalla, D. Catalano, C. Chevalier, and D. Pointcheval. Efficient two-party password-based key exchange protocols in the UC framework. In *CT-RSA 2008*, *LNCS* 4964, pages 335–351. Springer, Heidelberg, April 2008. (Pages 13 and 27.)

ACP09. M. Abdalla, C. Chevalier, and D. Pointcheval. Smooth projective hashing for conditionally extractable commitments. In *CRYPTO 2009*, *LNCS* 5677, pages 671–689. Springer, Heidelberg, August 2009. (Page 27.)

AP05. M. Abdalla and D. Pointcheval. Simple password-based encrypted key exchange protocols. In *CT-RSA 2005*, *LNCS* 3376, pages 191–208. Springer, Heidelberg, February 2005. (Page 22.)

BBDV16. J. Blocki, M. Blum, A. Datta, and S. Vempala. Towards human computable passwords. *arXiv preprint arXiv:1404.0024v4*, 2016. (Pages 3, 4, 5, 16, 17, 18, 19, and 20.)

BC08. J. Bringer and H. Chabanne. Trusted-HB: a low-cost version of HB+ secure against man-in-the-middle attacks. Cryptology ePrint Archive, Report 2008/042, 2008. http://eprint.iacr.org/2008/042. (Page 5.)

BCD06. J. Bringer, H. Chabanne, and E. Dottax. HB++: a lightweight authentication protocol secure against some attacks. In *Security, Privacy and Trust in Pervasive and Ubiquitous Computing, 2006. SecPerU 2006. Second International Workshop on*, pages 28–33. IEEE, 2006. (Page 5.)

BCVO12. R. Biddle, S. Chiasson, and P. Van Oorschot. Graphical passwords: Learning from the first twelve years. *ACM Comput. Surv.*, 44(4):19:1–19:41, September 2012. (Page 5.)

BM92. S. M. Bellovin and M. 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. (Pages 1, 5, 13, 20, and 27.)

BN00. M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT 2000*, *LNCS* 1976, pages 531–545. Springer, Heidelberg, December 2000. (Page 13.)

BNPS03. M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185–215, June 2003. (Pages 4, 10, and 11.)

BPR00. M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *EUROCRYPT 2000*, *LNCS* 1807, pages 139–155. Springer, Heidelberg, May 2000. (Pages 2, 5, 6, and 8.)

BR93. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS 93*, pages 62–73. ACM Press, November 1993. (Pages 12 and 26.)

Can00. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. http://eprint.iacr.org/2000/067. (Page 26.)

Can01. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. (Page 2.)

CF01. R. Canetti and M. Fischlin. Universally composable commitments. In *CRYPTO 2001*, *LNCS* 2139, pages 19–40. Springer, Heidelberg, August 2001. (Page 26.)

CHK+05. R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. D. MacKenzie. Universally composable password-based key exchange. In *EUROCRYPT 2005*, *LNCS* 3494, pages 404–421. Springer, Heidelberg, May 2005. (Pages 13 and 27.)

CPS08. J.-S. Coron, J. Patarin, and Y. Seurin. The random oracle model and the ideal cipher model are equivalent. In *CRYPTO 2008*, *LNCS* 5157, pages 1–20. Springer, Heidelberg, August 2008. (Page 20.)

DP00. R. Dhamija and A. Perrig. Déjà vu: A user study using images for authentication. In *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9*, SSYM'00, pages 4–4, Berkeley, CA, USA, 2000. USENIX Association. (Page 5.)

DS16. Y. Dai and J. P. Steinberger. Indifferentiability of 8-round feistel networks. In *CRYPTO 2016, Part I*, *LNCS* 9814, pages 95–120. Springer, Heidelberg, August 2016. (Page 20.)

Dzi10. S. Dziembowski. How to pair with a human. In *SCN 10*, *LNCS* 6280, pages 200–218. Springer, Heidelberg, September 2010. (Page 5.)

FLM11. M. Fischlin, B. Libert, and M. Manulis. Non-interactive and re-usable universally composable string commitments with adaptive security. In *ASIACRYPT 2011*, *LNCS* 7073, pages 468–485. Springer, Heidelberg, December 2011. (Page 26.)

FPV15a. V. Feldman, W. Perkins, and S. Vempala. On the complexity of random satisfiability problems with planted solutions. In *47th ACM STOC*, pages 77–86. ACM Press, June 2015. (Pages 4 and 17.)

FPV15b. V. Feldman, W. Perkins, and S. Vempala. On the complexity of random satisfiability problems with planted solutions. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 77–86. ACM, 2015. (Page 18.)

Goo. Google Authenticator. Google, Inc. https://support.google.com/accounts/answer/1066447?hl=en&rd=1. (Page 5.)

Hal95. N. Haller. The s/key one-time password system. RFC 1760, IETF, February 1995. http://tools.ietf.org/html/rfc1760. (Page 5.)

HB01. N. J. Hopper and M. Blum. Secure human identification protocols. In *ASIACRYPT 2001*, *LNCS* 2248, pages 52–66. Springer, Heidelberg, December 2001. (Page 5.)

HKT11. T. Holenstein, R. Künzler, and S. Tessaro. The equivalence of the random oracle model and the ideal cipher model, revisited. In *43rd ACM STOC*, pages 89–98. ACM Press, June 2011. (Page 20.)

HMNS98. N. Haller, C. Metz, P. Nesser, and M. Straw. A one-time password system. RFC 2289, IETF, February 1998. http://tools.ietf.org/html/rfc2289. (Page 5.)

JMM+99. I. Jermyn, A. Mayer, F. Monrose, M. K. Reiter, and A. D. Rubin. The design and analysis of graphical passwords. In *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*, SSYM'99, pages 1–1, Berkeley, CA, USA, 1999. USENIX Association. (Page 5.)

JW05. A. Juels and S. A. Weis. Authenticating pervasive devices with human protocols. In *CRYPTO 2005*, *LNCS* 3621, pages 293–308. Springer, Heidelberg, August 2005. (Page 5.)

Kho14. K. A. Khoureich. hHB: a harder HB+ protocol. Cryptology ePrint Archive, Report 2014/562, 2014. http://eprint.iacr.org/2014/562. (Page 5.)

Kho15. K. A. Khoureich. Light-hHB: A new version of hHB with improved session key exchange. Cryptology ePrint Archive, Report 2015/713, 2015. http://eprint.iacr.org/2015/713. (Page 5.)

KI96. T. Katoh and H. Imai. An application of visual secret sharing scheme concealing plural secret images to human identification scheme. In *Proc. of SITA*, pages 661–664, 1996. (Page 5.)

KPZ98. M.-R. Kim, J.-H. Park, and Y. Zheng. Human-machine identification using visual cryptography. In *Proceedings of the 6th IEEE International Workshop on Intelligent Signal Processing and Communication Systems*, pages 178–182, 1998. (Page 5.)

LMM08. X. Leng, K. Mayes, and K. Markantonakis. HB-MP+ protocol: An improvement on the HB-MP protocol. In *RFID, 2008 IEEE International Conference on*, pages 118–124. IEEE, 2008. (Page 5.)

LT99. X.-Y. Li and S.-H. Teng. Practical human-machine identification over insecure channels. *Journal of Combinatorial Optimization*, 3(4):347–361, 1999. (Page 5.)

Mat96. T. Matsumoto. Human-computer cryptography: An attempt. In *ACM CCS 96*, pages 68–75. ACM Press, March 1996. (Page 5.)

MBH+05. D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. HOTP: An HMAC-based one-time password algorithm. RFC 4226, IETF, December 2005. https://tools.ietf.org/html/rfc4226. (Page 5.)

MI91. T. Matsumoto and H. Imai. Human identification through insecure channel. In *EUROCRYPT'91*, *LNCS* 547, pages 409–421. Springer, Heidelberg, April 1991. (Page 5.)

MMPR11. D. M'Raihi, S. Machani, M. Pei, and J. Rydell. TOTP: Time-based one-time password algorithm. RFC 6238, IETF, May 2011. https://tools.ietf.org/html/rfc6238. (Page 5.)

MP07. J. Munilla and A. Peinado. HB-MP: A further step in the HB-family of lightweight authentication protocols. *Computer Networks*, 51(9):2262–2267, 2007. (Page 5.)

Nys07. M. Nystroem. The EAP protected one-time password protocol (EAP-POTP). RFC 4793, IETF, February 2007. http://tools.ietf.org/html/rfc4793. (Page 5.)

PS10. K. G. Paterson and D. Stebila. One-time-password-authenticated key exchange. In *ACISP 10*, *LNCS* 6168, pages 264–281. Springer, Heidelberg, July 2010. (Pages 5, 13, and 22.)

RBBK01. P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *ACM CCS 01*, pages 196–205. ACM Press, November 2001. (Page 13.)

RSA. RSA SecurId Hardware Tokens. RSA Security. https://www.rsa.com/en-us/products-services/identity-access-management/securid/hardware-tokens. (Pages 5, 12, and 21.)

TvO04. J. Thorpe and P. C. van Oorschot. Graphical dictionaries and the memorable space of graphical passwords. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 135–150. USENIX, 2004. (Page 5.)

vABHL03. L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *EUROCRYPT 2003*, *LNCS* 2656, pages 294–311. Springer, Heidelberg, May 2003. (Page 5.)
WHT95. C.-H. Wang, T. Hwang, and J.-J. Tsai. On the Matsumoto and Imai's human identification scheme. In *EUROCRYPT'95*, *LNCS* 921, pages 382–392. Springer, Heidelberg, May 1995. (Page 5.)
WK04. D. Weinshall and S. Kirkpatrick. Passwords you'll never forget, but can't recall. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '04, pages 1399–1402, New York, NY, USA, 2004. ACM. (Page 5.)

A Building Blocks

A.1 Commitment Scheme

We first recall the notation, and give more details about the advanced security notions.

COMMITMENT SCHEME: SYNTAX AND SECURITY. A (non-interactive) commitment scheme $\mathcal{CS}$ is defined by Setup that defines the global public parameters, and two other algorithms:

- $\mathsf{Com}(x)$: on input a message x, and some internal random coins, it outputs a commitment c together with an opening value s;
- $\mathsf{Open}(c, s)$: on input a commitment c and then opening value s, it outputs either the committed value x or $\perp$ in case of invalid opening value.

The **correctness** condition requires that for every x, if $(c, s) = \mathsf{Com}(x)$, then $\mathsf{Open}(c, s)$ outputs x. The usual **security notions** for commitment schemes are the *hiding* property, which says that x is hidden from c, and the *binding* property, which says that once c has been sent, no adversary can open it in more than one way. We respectively denote $\mathsf{Adv}^{\mathsf{hiding}}_{\mathcal{CS}}(\mathcal{A})$ (we will not actually use this one) and $\mathsf{Adv}^{\mathsf{binding}}_{\mathcal{CS}}(\mathcal{A})$ the advantages an adversary may get against these two notions.
For the additional security notions of *extractability* and *equivocality*, we need trapdoors, generated by an alternative setup algorithm and privately given to the simulator. And then, the hiding and binding properties are more delicate to be satisfied, hence the additional (probabilistic) algorithms: after a setup phase Setup' that defines the global public parameters available to everybody, with additional trapdoors we consider in the global private parameters, available to the simulator, we have

- $\mathsf{SimCom}()$, that takes as input the trapdoor and outputs a pair (c, eqk) where c is a fake commitment and eqk is the equivocation key;
- $\mathsf{OpenCom}(c, \mathsf{eqk}, m)$ that takes as input a fake commitment, its equivocation key and a message, and outputs an opening value s;
- $\mathsf{ExtCom}(c)$ that takes as input a non-fake commitment and outputs the message m it commits to.

These algorithms must first satisfy the two following properties:

- **Trapdoor Correctness**: (equivocality) for any message m, and any $(c, \mathsf{eqk}) \stackrel{\$}{\leftarrow} \mathsf{SimCom}()$ and $s \leftarrow \mathsf{OpenCom}(c, \mathsf{eqk}, m)$, we have $\mathsf{Open}(c, s) = m$; (extractability) for any message m and any $(c, s) \leftarrow \mathsf{Com}(m)$, we have $\mathsf{ExtCom}(c) = m$;
- **Setup Indistinguishability**: the official setup Setup and the new one Setup' generate indistinguishable global public parameters. We denote $\mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{A})$ the advantage an adversary $\mathcal{A}$ can get in distinguishing the global public parameters generated by the two setup phases.

In the trapdoor setting (when Setup' is used), the adversary is not given the trapdoors but just oracle access to the equivocation and extraction capabilities:

- $\mathsf{GenEquivCommit}()$ generates $(c, \mathsf{eqk}) \stackrel{\$}{\leftarrow} \mathsf{SimCom}()$, stores $(c, \mathsf{eqk}) \in \Psi$, and outputs c;
- $\mathsf{OpenEquivCommit}(c, m)$ first looks whether $c \in \Omega$ in which case it outputs $\perp$, otherwise it searches for $(c, \cdot) \in \Psi$, retrieves the matching eqk, stores $c \in \Omega$, and outputs $s \leftarrow \mathsf{OpenCom}(c, \mathsf{eqk}, m)$. It outputs $\perp$ if no (c, eqk) was found in Ψ;
- $\mathsf{ExtractCommit}(c)$ first looks whether $(c, \cdot) \in \Psi$ in which case it outputs $\perp$, otherwise it outputs $m \leftarrow \mathsf{ExtCom}(c)$.

The list Ψ is to keep track of the fake commitments, to exclude extraction on them, and the list Ω is to guarantee one opening only for any fake commitment. And then, with unlimited access to these oracles, we still expect the hiding and the binding properties to hold. They can be more formally modeled by the two following properties that, together with the setup indistinguishability, imply both the basic hiding and binding properties (see [ABB+13] for more details):

– **(Strong) Commitment Equivocality Indistinguishability**: the real commitment algorithms Com/Open and the fake-commitment algorithms SimCom/OpenCom generate indistinguishable commitments and opening values. For any adversary $\mathcal{A}$, we denote $\mathsf{Adv}^{\text{s-eq}}_{\mathcal{CS}}(\mathcal{A})$ its advantage in distinguishing the commitments and opening values generated by the two kinds of algorithms (even with unlimited access to the oracles GenEquivCommit, OpenEquivCommit, and ExtractCommit).
– **(Strong) Binding Extractability**: one cannot fool the extractor, *i.e.* produce a commitment c and a valid opening s to a message $m \neq \mathsf{ExtCom}(c)$. For any adversary $\mathcal{A}$, we denote $\mathsf{Adv}^{\text{s-binding}}_{\mathcal{CS}}(\mathcal{A})$ its advantage in generating a commitment c that it can open in a different way than the extraction algorithm (even with unlimited access tothe oracles GenEquivCommit, OpenEquivCommit, and ExtractCommit).

When a commitment scheme satisfies setup indistinguishability, strong commitment equivocality indistinguishability, and strong binding extractability, which additionally imply the basic hiding and binding properties, we say it is *strongly secure*.

Commitment Scheme: Instantiation. As shown in [ABB+13], UC-secure commitment schemes [CF01, Can00] are strongly secure, and so are enough for us. As a consequence, the UC-secure non-interactive constructions from [FLM11, ABB+13] fulfill all our requirements, in the standard model, with negligible advantages for any adversary, under the Decisional Diffie-Hellman assumption. However they will not be efficient enough for our purpose. On the other hand, the simple commitment scheme that commits m with large enough random coins r into $c = \mathcal{H}(m, r)$ is quite efficient and also fulfill all the above requirements, in the random oracle model [BR93]: Given a hash function $\mathcal{H}$ onto $\{0,1\}^\lambda$, the commitment scheme is defined as follows:

– $\mathsf{Com}(m)$: Generate $r \xleftarrow{\$} \{0,1\}^{2\lambda}$ and output $(c \leftarrow \mathcal{H}(m, r), s \leftarrow (m, r))$;
– $\mathsf{Open}(c, s = (m, r))$: if $\mathcal{H}(s) = c$, return m, otherwise, return $\perp$.

In the random oracle model, this simple scheme is trivially computationally *binding* ($\mathcal{H}$ is collision-resistant) and statistically *hiding* (for a large $r \in \{0,1\}^{2\lambda}$, there are almost the same number of possible r —actually, 2^λ— for any m, that would lead to the commitment c) in the ROM.
Additionally, we can use the *programmability* of the random oracle for *equivocality* and the list of query-answer for *extractability*:

– $\mathsf{SimCom}()$: Return $c \xleftarrow{\$} \{0,1\}^\lambda$.
– $\mathsf{OpenCom}(c, \cdot, m)$: Generate $r \xleftarrow{\$} \{0,1\}^{2\lambda}$, set $\mathcal{H}(m, r) \leftarrow c$, and return r.
– $\mathsf{ExtCom}(c)$: Search in list of queries to $\mathcal{H}$ which was output to c and return the first corresponding m.

By construction, we have the equivocality correctness, unless the value $\mathcal{H}(m, r)$ has already been asked, which is quite unlikely, since r is a fresh random. Extractability correctness is also ensured, as the recovered value m effectively commits to c. We also have perfect *setup indistinguishability*, so $\mathsf{Adv}^{\text{setup-ind}}_{\mathcal{H}}(\mathcal{A}) = 0$ for any adversary $\mathcal{A}$. The only way to distinguish a fake commitment from a real commitment would be to ask (m, r) to the oracle before OpenCom, hence to guess r. Therefore $\mathsf{Adv}^{\text{s-eq}}_{\mathcal{H}}(\mathcal{A}) \leq q_{\mathcal{H}} \times 2^{-2\lambda}$, for any adversary $\mathcal{A}$ asking at most $q_{\mathcal{H}}$ oracle queries and the scheme has *strong commitment equivocability indistinguishability*.
Moreover, this scheme has *strong binding extractability*. Suppose an adversary $\mathcal{A}$ is able to produce a tuple (c, s) that breaks the strong binding extractability. Hence if $(m', r') \leftarrow \mathsf{ExtCom}(c)$, $\mathcal{H}(m, r) = \mathcal{H}(m', r')$. This means that there was a collision between true random values, which is bounded by the birthday paradox. Hence: $\mathsf{Adv}^{\text{s-binding}}_{\mathcal{H}}(\mathcal{A}) < q_{\mathcal{H}}^2 \times 2^{-\lambda}$.
Therefore, this commitment scheme is *strongly secure* in the random oracle model.

The functionality $\mathcal{F}_{\mathsf{pake}}$ is parameterized by a security parameter k. It interacts with an adversary $\mathcal{S}$ and a set of parties $P_1, \ldots, P_n$ via the following queries:

- **Upon receiving a query $(\texttt{NewSession}, \mathsf{sid}, P_i, P_j, \mathsf{pw})$ from party P_i:**
  Send $(\texttt{NewSession}, \mathsf{sid}, P_i, P_j)$ to $\mathcal{S}$. If this is the first `NewSession` query, or if this is the second `NewSession` query and there is a record $(\mathsf{sid}, P_j, P_i, \mathsf{pw}')$, then record $(\mathsf{sid}, P_i, P_j, \mathsf{pw})$ and mark this record `fresh`.
- **Upon receiving a query $(\texttt{TestPwd}, \mathsf{sid}, P_i, \mathsf{pw}')$ from the adversary $\mathcal{S}$:**
  If there is a record of the form (P_i, P_j, pw) which is `fresh`, then do: If $\mathsf{pw} = \mathsf{pw}'$, mark the record `compromised` and reply to $\mathcal{S}$ with "correct guess". If $\mathsf{pw} \neq \mathsf{pw}'$, mark the record `interrupted` and reply with "wrong guess".
- **Upon receiving a query $(\texttt{NewKey}, \mathsf{sid}, P_i, \mathsf{sk})$ from the adversary $\mathcal{S}$:**
  If there is a record of the form $(\mathsf{sid}, P_i, P_j, \mathsf{pw})$, and this is the first `NewKey` query for P_i, then:
  - If this record is `compromised`, or either P_i or P_j is corrupted, then output $(\mathsf{sid}, \mathsf{sk})$ to player P_i.
  - If this record is `fresh`, and there is a record (P_j, P_i, pw') with $\mathsf{pw}' = \mathsf{pw}$, and a key sk' was sent to P_j, and (P_j, P_i, pw) was `fresh` at the time, then output $(\mathsf{sid}, \mathsf{sk}')$ to P_i.
  - In any other case, pick a new random key sk' of length k and send $(\mathsf{sid}, \mathsf{sk}')$ to P_i.

  Either way, mark the record $(\mathsf{sid}, P_i, P_j, \mathsf{pw})$ as `completed`.

Figure 5. Ideal Functionality $\mathcal{F}_{\mathsf{pake}}$ for PAKE

A.2 Password-Authenticated Key Exchange

A PAKE protocol is an interactive protocol between two parties who share a common low entropy secret (a password). At the end of the protocol, the parties output a session key, a session id and partner id. The correctness requires that any honest PAKE execution results in the parties outputting the same session key and session id, and the partner id being the identity of the other party.

The goal of the protocol is to guarantee privacy and (implicit) authentication of the session key in the presence of an attacker. Our proofs will use a UC-secure PAKE, whose ideal functionality $\mathcal{F}_{\mathsf{pake}}$ is presented on Figure 5, taken from [CHK+05].

The main idea is the following: If neither party is corrupted and the adversary does not attempt any password guess, then the two players both end up with either the same uniformly-distributed session key if the passwords are the same, or uniformly-distributed independent session keys if the passwords are distinct. However, if one party is corrupted (the adversary was given the password), or if the adversary successfully guessed the player's password (the session is then marked as `compromised`), the adversary is granted the right to fully determine its session key. In case of wrong guess (the session is then marked as `interrupted`), the two players are given independently-chosen random keys. A session that is nor `compromised` nor `interrupted` is called `fresh`, which is its initial status.

In the UC-framework, the security of a concrete protocol is proven by exhibiting a simulator $\mathcal{S}$ such that, anything an adversary $\mathcal{A}$ could do against honest players in the real protocol could be achieved by the same adversary $\mathcal{A}$ against the ideal functionality $\mathcal{F}$, with the simulator $\mathcal{S}$ as an interface between $\mathcal{A}$ and $\mathcal{F}$. But the ideal functionality is secure, by definition, and the combination of $\mathcal{S}$ and $\mathcal{A}$ cannot do anything harmful against the honest players using $\mathcal{F}$. As a consequence, $\mathcal{A}$ cannot do anything harmful against the honest players in the real protocol execution.

The security of a UC-PAKE PAKE is thus measured by the advantage a distinguisher $\mathcal{Z}$ could get in distinguishing the real world (the interactive protocol between honest players with an adversary $\mathcal{A}$) and the ideal world (the honest players directly dealing with the ideal functionality $\mathcal{F}_{\mathsf{pake}}$, while the simulator $\mathcal{S}$ makes the interface with the adversary $\mathcal{A}$): $\mathsf{Adv}^{\mathsf{pake}}_{\mathsf{PAKE}}(\mathcal{S}, \mathcal{A}, \mathcal{Z})$ thus denotes the advantage the distinguisher $\mathcal{Z}$ can get in distinguishing the two worlds.

When using a UC-PAKE in black-box, we will also assume the existence of a simulator $\mathcal{S}$ which makes this advantage negligible for any adversary $\mathcal{A}$, and any distinguisher $\mathcal{Z}$, for the ideal functionality $\mathcal{F}_{\mathsf{pake}}$ recalled on Figure 5.

Note that the classical EKE [BM92] protocol that encrypts a Diffie-Hellman key exchange, using the password as encryption key, is UC-secure [ACCP08], under the Computational Diffie-Hellman assumption in the ideal-cipher model. In addition, it is quite efficient. But other constructions also exist in the standard model [CHK+05, ACP09], under the Decisional Diffie-Hellman assumption.

B Simplified Basic HAKE Construction

In section 6.1, we explained that with a device, since the secrecy of the password must be maintained only temporarily, we can lessen the requirement on the PAKE in order to achieve a better efficiency. Taking advantage of this, we present, on Figure 6, the simplified basic HAKE, that is very similar to the basic generic HAKE of Section 4.2. As said above, it does not use a full PAKE, but just commitments to mutually check the knowledge of the ephemeral secret before it is revealed, which is enough in this setting.

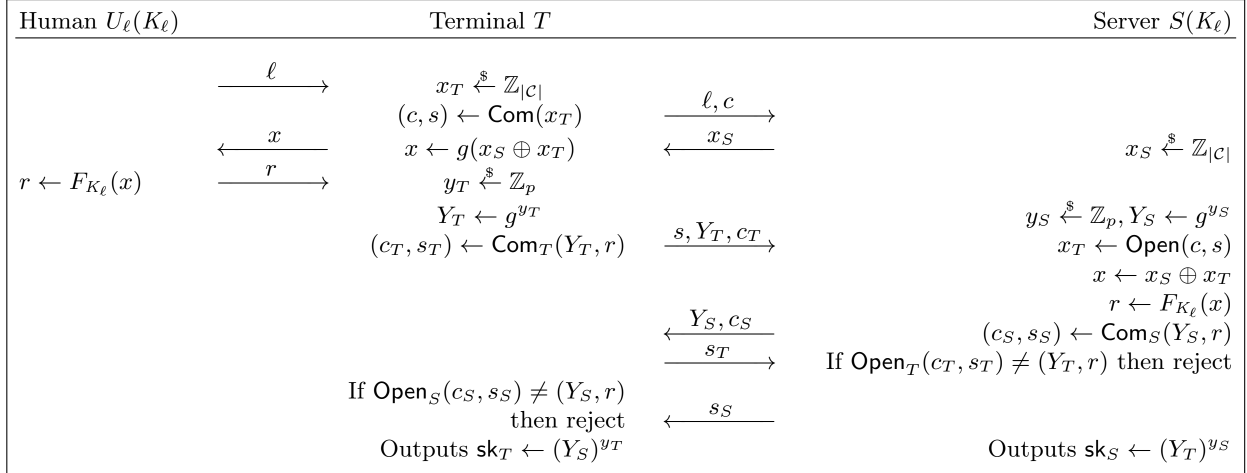

Figure 6. Simplified Basic HAKE Construction

We do not provide a proof, but it is easy to see that the hiding/binding properties of the commitment scheme replace the security properties of the PAKE in the proof of the basic HAKE in Appendix C. The only change from Theorem 3 would be that q_r is now only less or equal to $n_\mathsf{total} = n_\mathsf{comp} + n_\mathsf{uncomp} + n_\mathsf{serv}$ (the total number of sessions), which is not an issue for a device-assisted HC function.

C Proof of Theorem 3

We recall the protocol on Figure 2, and provide a more precise version of Theorem 3.

Theorem 3. *Consider the Basic HAKE protocol defined in Figure 2. Let $\mathcal{A}$ be an adversary against the privacy security game with static compromises, running within a time bound t and using less than n_comp compromised terminal sessions, n_uncomp uncompromised terminal sessions, n_serv server sessions and $n_\mathsf{active} \leq n_\mathsf{comp} + n_\mathsf{uncomp} + n_\mathsf{serv}$ active sessions. Then there exist an adversary $\mathcal{B}_4$ attacking the 2-party n_comp-unforgeability of the HC function family with $q_\mathsf{r}, q_\mathsf{c}, q_\mathsf{t}$ queries of the corresponding type, an adversary $\mathcal{A}$ and a distinguisher $\mathcal{B}_3$ attacking UC-security of the PAKE with a simulator $\mathcal{S}_\mathsf{pake}$ as well as three adversaries $\mathcal{D}_1$, $\mathcal{D}_2$, and $\mathcal{B}'_2$ against the commitment scheme properties, all running in time t, such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq\ & \mathsf{Adv}^{n_\mathsf{comp}\text{-}\mathsf{uf}}_{F}(\mathcal{B}_4) + 2 \times \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{PAKE}}(\mathcal{S}_\mathsf{pake}, \mathcal{A}, \mathcal{B}_3) \\ & + 2 \times \left(\mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{D}_1) + \mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{D}_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{B}'_2)\right) ,\end{aligned}$$

where $q_\mathsf{r} \leq n_\mathsf{comp}$, $q_\mathsf{t} \leq n_\mathsf{active}$, and $q_\mathsf{c} \leq n_\mathsf{uncomp} + n_\mathsf{comp} + n_\mathsf{serv}$.

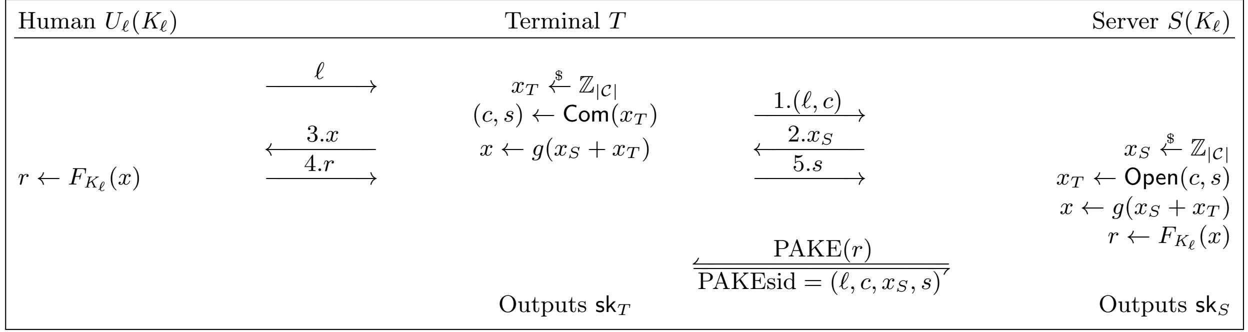

Figure 2. Basic Generic HAKE Construction (reminder)

Proof. We define a serie of games that aims at bounding the privacy advantage of $\mathcal{A}$. We denote $\mathcal{B}_i$ the simulator for game $\mathbf{G}_i$, that outputs 1 if $b \leftarrow \mathcal{A}$ ($\mathcal{A}$ wins in guessing b) and 0 otherwise. We also define $\mathcal{D}_i$ the distinguisher between games $\mathbf{G}_i$ and $\mathbf{G}_{i-1}$, for $i > 1$. In general, the distinguisher $\mathcal{D}_i$ between the two successive games exactly behaves as the simulator in $\mathbf{G}_{i-1}$ with all the secrets, but just interactions with two distributions to distinguish (either on sets or on oracles). The distributions on the outputs are thus as close as the input distributions which are close enough under a computational assumption. We start from $\mathbf{G}_0$, the privacy security game, between the adversary $\mathcal{A}$ and the challenger. We re-write it below, with explicit definitions of the queries. Then, in the last game, we explain how a simulator does without knowing anymore the long-term secret keys but using instead the oracles `GetResp`, `GetRandChal`, and `TestResp` to replace calls to F_{K_ℓ}.
We stress that the compromises are static, which means that `Compromise`-queries must happen before any other flow in a terminal session. However, infections can still be adaptively made (but on compromised sessions only).
We also require the internal PAKE primitive (denoted by the double-arrow on the Figure 2) to be UC-secure (see Figure 5), as we will need the ideal functionality $\mathcal{F}_{\mathsf{pake}}$ and the simulator $\mathcal{S}_{\mathsf{pake}}$ in the proof. In particular, we need to be able to simulate transcripts between honest players, without knowing the password, and we will need to be able to extract the password tried by the adversary. On the other hand, we will have to be able to simulate the answers to the `TestPwd`-queries.

Game $\mathbf{G}_0$: In this game, the simulator generates the public pamaters for the HC function, the commitment, and the PAKE. It also knows all the long-term secret keys of the users, which allows it to simulate every oracle π_P^j, as the latter would do in the real protocol, with a random bit b:

1. `SendServ`$(k, 1.(\ell, c))$: generate and send $x_S \xleftarrow{\$} \mathbb{Z}_{|\mathcal{C}|}$
2. `SendServ`$(k, 5.s)$: open the value $x_T \leftarrow \mathsf{Open}(c, s)$, set $x \leftarrow g(x_S + x_T)$, and compute $r \leftarrow F_{K_\ell}(x)$
3. `SendServ`-PAKE queries to π_S^k: run the PAKE protocol on r
4. `SendTerm`$(j, $`Start`$(\ell))$:
   (a) generate $x_T \xleftarrow{\$} \mathbb{Z}_{|\mathcal{C}|}$, commit $(c, s) \leftarrow \mathsf{Com}(x_T)$, and send c
   (b) If *compromised*, reveal x_T together with the random coins of the commitment
5. `SendTerm`$(j, 2.x_S)$:
   (a) Set $x \leftarrow g(x_S + x_T)$, compute $r \leftarrow F_{K_\ell}(x)$, and send the opening value s
   (b) If *compromised*, reveal r
6. `SendTerm`-PAKE queries to π_T^j: run the PAKE protocol on r
7. `SendHum`$(j, 3.x)$ (from an *infected* terminal π_T^j): Compute and return $r \leftarrow F_{K_\ell}(x)$
8. `Compromise`(j, ℓ): Unless a `Start-SendTerm`-query has already been sent to π_T^j, mark the instance π_T^j as *compromised* and reveal the random tape.
9. `Infect`(j): mark the session as *infected*, allowing `SendHum`
10. `Test`(j, P): according to b, and whether π_P^i is fresh or not, the real key sk_P, or a random key, or $\perp$ is returned.

By definition, $\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) = 2 \times \Pr_{\mathbf{G}_0}[b \leftarrow \mathcal{A}] - 1$.

Game $\mathbf{G}_1$: We now replace the Setup algorithm of the commitment by Setup', allowing equivocal commitments and extractability, but without any additional change: $|\Pr_{\mathbf{G}_1}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_0}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{D}_1)$, where $\mathcal{D}_1$ behaves as $\mathcal{B}_0$, with either Setup (which is $\mathbf{G}_0$) or Setup' (which is $\mathbf{G}_1$).

Game $\mathbf{G}_2$: We can now enforce random challenges, by extracting the adversary commitments and generating a commitment on a value that complement appropriately.

In the simulation, several steps will depend on whether the input message is oracle-generated, meaning it is the output of another oracle and whether the terminal (be it the source of an oracle-generated message or the local oracle) is compromised or not.

In the case of `SendTerm`-queries, we will use the terminology *compromised* or *uncompromised* to denote the fact that the local terminal instance was compromised or not, as well as *oracle-generated* or *non-oracle-generated* to denote the fact the input message was generated as output by a server oracle (from a `SendServ`-query).

In the case of `SendServ`-queries, the terminology *remote-compromised* or *remote-uncompromised* will denote the fact that the input message was generated as output by a terminal oracle (from a `SendTerm`-query) that is compromised or not. However, when the input message does not come as an output of a server oracle it is called *non-oracle-generated.* Therefore there are three cases : *remote-compromised*, *remote-uncompromised*, and *non-oracle-generated.*

In order to enforce random challenges, in step 1, when c is *non-oracle-generated* or *remote-compromised* (the terminal is compromised or infected), we extract x_T and generate $x_S \leftarrow g^{-1}(x) + x_T$ so that $g(x_T + x_S)$ is the expected random challenge; in step 4, when the terminal is not *compromised*, it generates a fake commitment, and in step 5 it opens it to x_T that complement correctly with x_S.

If the enforced random challenge does not get the expected value in step 2, the simulator $\mathcal{B}_2'$ outputs (c, s) that breaks the *strong binding extractability.* Moreover, the *strong simulation indistinguishability* ensures that otherwise it is hard to distinguish this game from the previous one, by defining $\mathcal{D}_2$ exactly as the simulator $\mathcal{B}_1$, using either $\mathsf{Com}/\mathsf{Open}$ (which is $\mathbf{G}_1$) or $\mathsf{SimCom}/\mathsf{OpenCom}$ (which is $\mathbf{G}_2$).

Hence, under the strong-security of the commitment scheme, the simulation remains the same to the adversary:

$$|\Pr_{\mathbf{G}_2}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_1}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{D}_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{B}_2')$$

Game $\mathbf{G}_3$: We now use $\mathcal{S}_{\mathsf{pake}}$ to emulate all the messages our simulator should send for the PAKE protocol: $|\Pr_{\mathbf{G}_3}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_2}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{PAKE}}(\mathcal{S}_{\mathsf{pake}}, \mathcal{A}, \mathcal{B}_3)$.

Since we are using a UC-secure PAKE, unless that adversary has guessed the password r, it has no information about the session key. As a consequence, unless the adversary has asked a successful `TestPwd`-query (event `GuessedPwd`), it has no advantage in breaking the privacy of our HAKE: $2 \times \Pr_{\mathbf{G}_3}[b \leftarrow \mathcal{A}] - 1 \leq \Pr_{\mathbf{G}_3}[\mathtt{GuessedPwd}]$.

As a consequence,

$$\begin{aligned}\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq \Pr_{\mathbf{G}_3}[\mathtt{GuessedPwd}] &+ 2 \times \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{PAKE}}(\mathcal{S}_{\mathsf{pake}}, \mathcal{A}, \mathcal{B}_3). \\ &+ 2 \times \left(\mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{D}_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{B}_2') + \mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{D}_1)\right)\end{aligned}$$

Game $\mathbf{G}_4$: Eventually, we can make use of oracles `GetResp`, `GetRandChal`, and `TestResp` to replace calls to F_{K_ℓ}: we do not know anymore the long-term keys of the users, and we focus on a user $U = U_\ell$ (all the other calls can be done as above), but we know the trapdoor for the commitment scheme.

Thanks to the extractable and equivocal commitment scheme, we can inject random challenges and use the adversary to break the HC function on one of them. The simulator will indeed be able to set x of its choice (from the HC function random selection) for either the server oracle or the terminal oracle, while the adversary has to commit its share and cannot equivocate.

Let us now give the details of the final game: the simulator uses the simulator $\mathcal{S}_{\mathsf{pake}}$ to generate the public parameters for the PAKE, generates the public parameters for the commitment with the trapdoors (for extraction and equivocality), and uses the challenge instance of the HC function family for the parameters of F. It also set Λ to an empty set. It will be used to keep track of the known challenge-responses. Then it answers the oracle queries as follows:

1. $\texttt{SendServ}(k, 1.(\ell, c))$:
   - If *remote-uncompromised*: store $b'_x \leftarrow 1$, generate $x_S \xleftarrow{\$} \mathbb{Z}_{|\mathcal{C}|}$ and send it
   - If *remote-compromised* or *non-oracle-generated*:
     (a) Store $b'_x \leftarrow 0$
     (b) Generate $x \leftarrow \texttt{GetRandChal}(0)$.
     (c) Using the extraction key of the commitment scheme, open c to learn x_T
     (d) Compute and send $x_S \leftarrow g^{-1}(x) + x_T$
2. $\texttt{SendServ}(k, 5.s)$: Do nothing (since the committed value x_T is already known)
3. $\texttt{SendServ}$-PAKE queries to π_S^k:
   - If $(x, r) \in \Lambda$ for some r[3]: run the PAKE protocol on r and issue a $\texttt{TestResp}(r, b'_x)$.
   - Otherwise: use the simulator $\mathcal{S}_{\mathsf{pake}}$ to generate the server flows (for an accepting PAKE transcript if the flows are *oracle-generated*). In case of a $\texttt{TestPwd}$-query on a candidate $r_{\mathcal{A}}$, run $\texttt{TestResp}(r_{\mathcal{A}}, b'_x)$
4. $\texttt{SendTerm}(j, \texttt{Start}(\ell))$: Initialize $b_x \leftarrow 0$ and
   - If *uncompromised*, generate and send an equivocal commitment c
   - If *compromised*:
     (a) generate $x_T \xleftarrow{\$} \mathbb{Z}_{|\mathcal{C}|}$, commit $(c, s) \leftarrow \mathsf{Com}(x_T)$, and send c
     (b) reveal x_T together with the random coins of the commitment
5. $\texttt{SendTerm}(j, 2.x_S)$:
   - If *compromised*, set $x^* \leftarrow g(x_S + x_T)$. Compute $r \leftarrow \texttt{GetResp}(x^*)$[4]. Then, store (x^*, r) in Λ, send the opening value s and reveal r
   - If *uncompromised*:
     (a) Set $x \leftarrow \texttt{GetRandChal}(1), x_T \xleftarrow{\$} x_S + g^{-1}(x)$ and $b_x \leftarrow 1$
     (b) Generate and send s such that $\mathsf{Open}(c, s) = x_T$, using the equivocation key of the commitment
6. $\texttt{SendTerm}$-PAKE queries to π_T^j:
   - If *compromised*, run the PAKE protocol on the known r
   - If *uncompromised* and *oracle-generated*, use the simulator $\mathcal{S}_{\mathsf{pake}}$ to generate the terminal flow (for an accepting PAKE transcript if the flows remain oracle-generated)
   - If *uncompromised* and *non-oracle-generated*, use the simulator $\mathcal{S}_{\mathsf{pake}}$ to generate the terminal flow. In case of a $\texttt{TestPwd}$-query on a candidate $r_{\mathcal{A}}$, run $\texttt{TestResp}(r_{\mathcal{A}}, b_x)$
7. $\texttt{SendHum}(j, 3.x)$ (from an *infected* terminal):
   - If $(x, r) \in \Lambda$ for some r, output r ;
   - Otherwise, compute $r \leftarrow \texttt{GetResp}(x)$, store (x, r) in Λ and output r.
8. $\texttt{Compromise}(j, \ell)$: (unchanged) Unless a $\texttt{Start}$-$\texttt{SendTerm}$-query has already been sent to π_T^j, mark the instance π_T^j as *compromised* and reveal the random tape.
9. $\texttt{Infect}(j)$: (unchanged) mark the session as *infected*, allowing $\texttt{SendHum}$
10. $\texttt{Test}(j, P)$: (unchanged) according to b, and whether π_P^i is fresh or not, the real key sk_P, or a random key, or $\perp$ is returned.

Note that the event $\texttt{GuessedPwd}$ now means that a $\texttt{TestResp}$-answer was positive, we no previous $\texttt{GetResp}$-query. This is exactly a success in the unforgeability of the HC function family, so

$$\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq \mathsf{Adv}^{n_{\mathsf{comp}}\text{-}\mathsf{uf}}_{F}(\mathcal{B}_4) + 2 \times \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{PAKE}}(\mathcal{S}_{\mathsf{pake}}, \mathcal{A}, \mathcal{B}_3). \\ + 2 \times \left(\mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{D}_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{B}'_2) + \mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{D}_1)\right)$$

[3] In particular, this can be the case if the $\texttt{SendServ}(k, 1.c)$-query was *remote-compromised* or if r was queried through the $\texttt{SendHum}$ query of an *infected* terminal session.

[4] If such a query is not allowed because too many were asked since the last $\texttt{GetRandChal}$, it first issues a $\texttt{GetRandChal}(0)$ query (and discards the result)

We now have to count how many queries are asked by our simulator $\mathcal{B}_4$ in game $\mathbf{G}_4$.
One can note from this simulation that any interaction between the adversary and a safe player (*uncompromised* terminal/server) results in an $x \in \Lambda_0$ (generated by a GetRandChal-query from the simulator on behalf of the safe player). Moreover,

- in each *passive session* with an *uncompromised* terminal, there is just a GetRandChal-query;
- for each server interacting with a *compromised* terminal, the simulator asks a GetRandChal-query, a GetResp-query and a correct TestResp, hence ctr overall stays the same;
- in each session between a *uncompromised* terminal and an adversary trying to impersonate the server (some non-oracle-generated flows), there might be a GetRandChal-query and a TestResp-query;
- in each session between a *compromised* terminal and an adversary trying to impersonate the server (some non-oracle-generated flows) with an x of its choice, there might be a GetResp-query (and a GetRandChal in some cases), but no TestResp. Hence, ctr is incremented.
- in each session with an *infected* terminal that directly queries the user on its own challenge, there is a GetResp-query. Hence ctr is incremented.
- in each session between an honest server and an adversary playing on behalf of the user/terminal (after *infecting* or not the terminal), there are a GetRandChal-query and a TestResp-query. If one of the two previous terminal session situations occured between those, the TestResp will decrement ctr[5] if the PAKE succeeds
  Or none occured, in which case π_S^k is *fresh*, but a success of the PAKE means winning the unforgeability game;

As a consequence,

- a GetResp-query only appears in sessions with a compromised terminal Hence, q_{r} is at most the number of *compromised* terminal sessions, that is
  $$q_{\mathsf{r}} \leq n_{\mathsf{comp}};$$
- a TestResp-query only appears in sessions between a safe player (uncompromised terminal or honest server) and the other being impersonated by the adversary. In particular, such session are active, hence q_{t} is at most the number of active sessions (n_{active}).
  $$q_{\mathsf{t}} \leq n_{\mathsf{active}};$$
- a GetRandChal-query only appears in sessions with a safe player (uncompromised terminal or honest server) or, in some cases, in sessions where a GetResp-query was issued. Hence, q_{c} is at most the sum of the number of *uncompromised* terminal sessions (n_{uncomp}), GetResp-queries and the number of server sessions with a *non-oracle-generated* flow 1.c ($n_{\mathsf{nog_c}}$),
  $$q_{\mathsf{c}} \leq n_{\mathsf{uncomp}} + q_{\mathsf{r}} + n_{\mathsf{nog_c}} \leq n_{\mathsf{uncomp}} + n_{\mathsf{comp}} + n_{\mathsf{serv}}.$$
- ctr is only incremented by *compromised* terminal session in which a GetResp-query was asked, while no concurrent server session succesfully terminated. Unfortunately, such a situation is not reliably detectable by the human, though it will play an important role for the Confirmed HAKE.
  $$\mathsf{ctr} \leq n_{\mathsf{comp}}$$

D Proof of Theorem 4

We recall the protocol on Figure 3 and associated theorem on the security level, that we then prove.

[5] Thus, counting both the terminal session that increments ctr and the server sessions that decrements it, ctr is unchanged

Human $U_\ell(K_\ell)$		Terminal T		Server $S(K_\ell)$
$\mathsf{accept} \leftarrow \mathtt{False}$				$\mathsf{accept} \leftarrow \mathtt{False}$
$\mathsf{terminate} \leftarrow \mathtt{False}$				$\mathsf{terminate} \leftarrow \mathtt{False}$
	$\xrightarrow{\ell}$	$x_T \xleftarrow{\$} \mathbb{Z}_{\lvert\mathcal{C}\rvert}$	$\xrightarrow{1.(\ell, c)}$	
	$\xleftarrow{3.x}$	$(c, s) \leftarrow \mathsf{Com}(x_T)$	$\xleftarrow{2.x_S}$	$x_S \xleftarrow{\$} \mathbb{Z}_{\lvert\mathcal{C}\rvert}$
$r \leftarrow F_{K_\ell}(x)$	$\xrightarrow{4.r}$	$x \leftarrow g(x_S + x_T)$	$\xrightarrow{5.s}$	$x_T \leftarrow \mathsf{Open}(c, s)$
$\mathsf{accept} \leftarrow \mathtt{True}$				$x \leftarrow g(x_S + x_T)$
			$\mathrm{PAKE}(r)$	$r \leftarrow F_{K_\ell}(x)$
			$\mathrm{PAKEsid} = (\ell, c, x_S, s)$	
		Parses key: $(k_T \Vert \mathsf{sk}_T)$		Parses key: $(k_S \Vert \mathsf{sk}_S)$
				$\mathsf{accept} \leftarrow \mathtt{True}$
$x_U \xleftarrow{\$} \mathcal{C}$	$\xrightarrow{4.x_U}$	$X_U \leftarrow \mathsf{Enc}_{k_T}(x_U)$	$\xrightarrow{7.X_U}$	$x_U \leftarrow \mathsf{Dec}_{k_S}(X_U)$
Verifies r_U	$\xleftarrow{9.r_U}$	$r_U \leftarrow \mathsf{Dec}_{k_T}(R_U)$	$\xleftarrow{8.R_U}$	$R_U \leftarrow \mathsf{Enc}_{k_S}(F_{K_\ell}(x_U))$
		Outputs sk_T		Outputs sk_S
$\mathsf{terminate} \leftarrow \mathtt{True}$	$\xrightarrow{0/1}$			$\mathsf{terminate} \leftarrow \mathtt{True}$

Figure 3. Confirmed HAKE Construction (reminder)

Theorem 4. *Consider the Confirmed HAKE protocol defined in Figure 3. Let $\mathcal{A}, \mathcal{A}'$ be adversaries against the privacy and authenticity security game of HAKE within a time bound t and using less than n_{comp}* compromised *terminal sessions, n_{uncomp}* uncompromised *terminal sessions, n_{serv} server sessions, $n_{\mathsf{active}} \leq n_{\mathsf{comp}} + n_{\mathsf{uncomp}} + n_{\mathsf{serv}}$ active sessions and n_{hr} human session that* reject *in the end. Then there exist two adversaries $\mathcal{B}_1, \mathcal{B}'_1$ attacking the 2-party $(n_{\mathsf{hr}} + 1)$-unforgeability of HC function family with $q_{\mathsf{r}}, q_{\mathsf{c}}, q_{\mathsf{t}}$ queries of the corresponding type, two adversaries $\mathcal{B}_2, \mathcal{B}'_2$ and two distinguishers $\mathcal{B}_3, \mathcal{B}'_3$ attacking UC-security of the PAKE with the simulator $\mathcal{S}_{\mathsf{pake}}$, two adversaries $\mathcal{B}_4, \mathcal{B}'_4$ against the authenticated encryption, as well as six adversaries $\mathcal{A}_1, \mathcal{A}'_1$, $\mathcal{A}_2, \mathcal{A}'_2$, and $\mathcal{A}_3, \mathcal{A}'_3$ against the commitment scheme properties, all running in time t, such that*

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq{}& \mathsf{Adv}^{(n_{\mathsf{hr}}+1)-\mathsf{uf}}_{F}(\mathcal{B}_1) + 2 \times \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{PAKE}}(\mathcal{S}_{\mathsf{pake}}, \mathcal{B}_2, \mathcal{B}_3) + 2 \times \mathsf{Adv}^{\mathsf{authenc}}_{\mathcal{ES}}(\mathcal{B}_4) \\
&+ 2 \times \left(\mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{A}_1) + \mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{A}_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{A}_3)\right), \\
\mathsf{Adv}^{\mathsf{auth}}_{\mathsf{HAKE}}(\mathcal{A}') \leq{}& \mathsf{Adv}^{(n_{\mathsf{hr}}+1)-\mathsf{uf}}_{F}(\mathcal{B}'_1) + 2 \times \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{PAKE}}(\mathcal{S}_{\mathsf{pake}}, \mathcal{B}'_2, \mathcal{B}'_3) + 2 \times \mathsf{Adv}^{\mathsf{authenc}}_{\mathcal{ES}}(\mathcal{B}'_4) \\
&+ 2 \times \left(\mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{A}'_1) + \mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{A}'_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{A}'_3)\right),
\end{aligned}$$

where $q_{\mathsf{r}} \leq 2n_{\mathsf{comp}}$, $q_{\mathsf{t}} \leq n_{\mathsf{active}}$, $q_{\mathsf{c}} \leq n_{\mathsf{comp}} + n_{\mathsf{uncomp}} + n_{\mathsf{serv}}$.

Proof (Privacy proof). The proof is very similar to the previous one, so we just introduce the new queries to add to the security games defined in Section 4.2, but only after the execution of the PAKE, and so the simulator (in the real game when it knows all the long-term keys) knows the output $(k_P \Vert \mathsf{sk}_P)$:

Game $\mathbf{G}_0$: In this game, we simply simulate every oracle according to the protocol, knowing the output of the PAKE, and all the honest players accept at the end of the PAKE.

1. Last `SendTerm`-PAKE-query (that led to the session key (k_T, sk_T)):
   - (a) generate $x_U \xleftarrow{\$} \mathcal{C}$ (on behalf of the user, and so the internal random coins stay secret even if π^j_T is *compromised*)
   - (b) send $X_U \leftarrow \mathsf{Enc}_{k_T}(X_U)$
2. `SendHum`$(j, 3.x)$ (from an *infected* terminal): Not only output $4.r$, as before, but additionally generate and send $x_U \xleftarrow{\$} \mathcal{C}$.
3. `SendServ`$(k, 7.X_U)$: Compute and send $R_U \leftarrow \mathsf{Enc}_{k_S}(F_{K_\ell}(\mathsf{Dec}_{k_S}(X_U)))$ (unless the decryption fails, `terminate` is set to `True`)

4. $\texttt{SendTerm}(j, 8.R_U)$: Compute $r'_U \leftarrow \mathsf{Dec}_{k_T}(R_U)$ and check whether $r'_U = F_{K_\ell}(x_U)$ or not (on behalf of the user). Unless the decryption fails, if the equality holds, the user accepts and the terminal sets $\mathsf{terminate}$ to $\mathtt{True}$
5. $\texttt{SendHum}(j, 9.r_U)$ (from an *infected* terminal): Accept if $r_U = F_{K_\ell}(x_U)$.

Game $\mathbf{G}_1$: In this game, we once again make use of the underlying HC security game oracles. But now, the simulator does not know anymore the long-term keys, and did not learn the output of the PAKE when the simulator $\mathcal{S}_{\mathsf{pake}}$ was involved.

1. Last $\texttt{SendTerm}$-PAKE-query:
   - If *compromised* (the simulator honestly ran the PAKE on behalf of the terminal, and so the simulator knows (k_T, sk_T), but the adversary too)
     (a) generate $x_U \leftarrow \texttt{GetRandChal}(1)$
     (b) send $X_U \leftarrow \mathsf{Enc}_{k_T}(x_U)$
   - If *uncompromised*, generate a random ciphertext X_U
2. $\texttt{SendHum}(j, 3.x)$ (from an *infected* terminal), as before, and additionally generate and send $x_U \leftarrow \texttt{GetRandChal}(1)$
3. $\texttt{SendServ}(k, 7.X_U)$:
   - If *remote-compromised*, the simulator knows (k_S, sk_S) and knows x_U; it then computes $r_U \leftarrow \texttt{GetResp}(x_U)$, stores (x_U, r_U) in Λ and sends $R_U \leftarrow \mathsf{Enc}_{k_0}(r_U)$
   - If *remote-uncompromised*, generate and send a random ciphertext R_U
   - If *non-oracle-generated*,
     - either the simulator knows k_S because of an honest execution of the PAKE with r. Then, it can get $x'_U \leftarrow \mathsf{Dec}_{k_S}(X_U)$. If this decrypts correctly, it checks if (x'_U, r'_U) is in Λ for some r'_U. If it is not, get $r'_U \leftarrow \texttt{GetResp}(x'_U)$. It then sends $\mathsf{Enc}_{k_S}(r'U)$, and set $\mathsf{terminate}$ to $\mathtt{True}$
     - or the simulator does not know k_S because it invoked the simulator $\mathcal{S}_{\mathsf{pake}}$, which means that the adversary should not know the session key either, and so is unable to generate a valid ciphertext: the simulator makes the server to abort.
4. $\texttt{SendTerm}(j, 8.R_U)$:
   - If *compromised* and *oracle-generated*, r_U is known and correct so run $\texttt{TestResp}(r_U, 1)$ and terminate
   - If *compromised* and *non-oracle-generated*, get $r'_U \leftarrow \mathsf{Dec}_{k_T}(R_U)$, and run $\texttt{TestResp}(r'_U, 1)$, to either terminate or reject (a decryption failure leads to a reject)
   - If *uncompromised* and *oracle-generated*, terminate
   - If *uncompromised* and *non-oracle-generated*, reject
5. $\texttt{SendHum}(j, 9.r_U)$ (from an *infected* terminal): Accept if $\texttt{TestResp}(r_U, 1)$.

Note that in the added flows, all $\texttt{GetRandChal}$ and $\texttt{TestResp}$ occur with bit 1, which was only used previously if the terminal oracle was *uncompromised*, in which case either the adversary issued a correct $\texttt{TestPwd}$, and the simulator already won the HC unforgeability game, or (k_P, sk_P) is unknown and the authenticated encryption hides all flows.

From the previous simulation, in case of passive sessions with a compromised terminal, in the first part of the simulation, the simulator honestly ran the PAKE, completing it with (k_T, sk_T). It can thus continue honestly if the adversary continues to forward oracle generated flows. It then learns a new challenge-response pair. If the adversary starts to play on behalf of the server, it has to generate a forgery for the HC function (checked by the $\texttt{TestResp}$-query).

We will define an adversary $\mathcal{B}'_1$ against the combined security notion for authenticated encryption.

In case of passive sessions with an uncompromised terminal, k_T is unknown to the adversary, then random ciphertexts can be sent, they are indistinguishable from real ciphertexts (from the semantic security of our secure authenticated encryption scheme, in which case $\mathcal{B}'_1$ is used as a distinguisher between the encryption of the real plaintext —as in game $\mathbf{G}_0$— or of a random plaintext —as in game $\mathbf{G}_1$—). Eventually, if the adversary tries to impersonate the server before the PAKE, and send non-oracle-generated messages to the terminal, they should not be encrypted under the correct key

k_T, unless the adversary has broken either the integrity of the authenticated encryption (in which case $\mathcal{B}'_1$ outputs the fake ciphertext message) or the PAKE (correct guess of r), which would have led to a positive answer to a `TestResp` during the simulation before: a forgery against the HC function.
As a consequence, unless the adversary has helped the simulator to break the unforgeability of the HC function family, this game $\mathbf{G}_1$ is indistinguishable from the previous one, and leads to accepted sessions by the human user for passive sessions only (with compromised terminals or not) with the expected server, which was our target.
We already showed that the server could only agree on a key (k_P, sk_P) with a terminal linked to the expected human user, hence the global security of our protocol: privacy and authentication. In addition, the two more flows allow the human user to detect whether the key sk_T is shared with the expected server, just leaking one more random challenge-response pair in case of compromised terminal.
With the same reasoning as in the previous proof, we have

$$\begin{aligned}\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq \mathsf{Adv}^{\eta-\mathsf{uf}}_{F}(\mathcal{B}_1) + 2 \times \mathsf{Adv}^{\mathsf{pake}}_{\mathsf{PAKE}}(\mathcal{S}_{\mathsf{pake}}, \mathcal{B}_2, \mathcal{B}_3) + 2 \times \mathsf{Adv}^{\mathsf{authenc}}_{\mathcal{ES}}(\mathcal{B}_4) \\ + 2 \times \left(\mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{A}_1) + \mathsf{Adv}^{\mathsf{s\text{-}eq}}_{\mathcal{CS}}(\mathcal{A}_2) + \mathsf{Adv}^{\mathsf{s\text{-}binding}}_{\mathcal{CS}}(\mathcal{A}_3)\right).\end{aligned}$$

However, there are a few changes regarding the number of queries involved in the simulation $\mathcal{B}_5$, but only on *compromised* sessions, as otherwise the simulator would not have played the PAKE honestly and therefore only random-looking messages are generated:

- There may be an additional `GetRandChal` in some *compromised* sessions.
- There may be an additional `TestResp`-query in some *active* sessions, but only if the PAKE was passive, without a previous `TestResp`-query. Otherwise k_T is random, and the decryption fails.
- There may be an additional `GetResp`-query in some server session, if the PAKE succeeded, which means a *compromised* terminal session must have been used or a win was alread reached.
- ctr may incremented, but only if the PAKE suceeded (meaning it was not durably increased by the previous flows). Moreover, if the human accepts, it is decreased (and overall stays the same).

Overall, it is notable that ctr is permanently increased only if the human rejects in the end or the HC unforgeability game is won. Hence:

$$\mathsf{ctr} \leq n_{\mathsf{hr}}$$

Moreover, ctr can never reach a value higher than its final value plus one. This means that $\eta = n_{\mathsf{hr}} + 1$ is sufficient for the HC unforgeability game.

Proof (Authenticity proof). The simulated game $\mathbf{G}_1$ is indistinguishable from the real one (either the privacy security game or the authentication security game). To break the server-authentication, the adversary should be able compute the answer to a random challenge x_U with no `GetResp` allowed. Breaking the user-authentication means that the adversary succeeded in the PAKE, in order to know k_S, and to send a valid ciphertext X_U and hence must also guess the answer r to a random x challenge. Since we exclude trivial attacks from the authentication security game, any winning strategy requires a successful `TestResp`-query. Hence, the advantage $\mathsf{Adv}^{\mathsf{auth}}(\mathcal{A})$ of the adversary in the authentication security game is upper-bounded the same way as $\mathsf{Adv}^{\mathsf{priv}}(\mathcal{A})$.

E Proof of Theorem 10

We recall the protocol on Figure 4, and provide a more precise version of Theorem 10.

Time	Human U_ℓ		Terminal T		Server S
$\leq t$	accept $\leftarrow$ False				accept $\leftarrow$ False
					terminate $\leftarrow$ False
t		$\xrightarrow{\ell}$	$x_T \xleftarrow{\$} \mathbb{Z}_p, X_T \leftarrow g^{x_T}$		$x_S \xleftarrow{\$} \mathbb{Z}_p, X_S \leftarrow g^{x_S}$
t		$\xrightarrow{\mathsf{pw}_t}$	$(c_T, s_T) \leftarrow \mathsf{Com}_T(X_T, \mathsf{pw}_t)$	$\xrightarrow{\ell, X_T, c_T}$	
t	accept $\leftarrow$ True			$\xleftarrow{X_S, c_S}$	$(c_S, s_S) \leftarrow \mathsf{Com}_S(X_S, \mathsf{pw}_t)$
$\vdots$			Wait for timeframe $> t$		Wait for timeframe $> t$
$> t$				$\xrightarrow{s_T}$	If $\mathsf{Open}_T(c_T, s_T) = (X_T, \mathsf{pw}_t)$ and
					$(\ell, t) \notin \Lambda$, store (ℓ, t) in Λ
$> t$					Otherwise reject
$> t$					accept $\leftarrow$ True
$> t$			Reject if	$\xleftarrow{s_S}$	Outputs $(X_T)^{x_S}$
			$\mathsf{Open}_S(c_S, s_S) \neq (X_S, \mathsf{pw}_t)$		
$> t$			Outputs $(X_S)^{x_T}$		terminate $\leftarrow$ True

Figure 4. Time-Based Device-Assisted HAKE Construction (reminder)

Theorem 10. *Consider the Time-Based Device-Assisted HAKE protocol defined in Figure 4. Let $\mathcal{A}, \mathcal{A}'$ be an adversaries against the privacy and user authentication security games with static compromises, running within time $t_\mathcal{A}$ and using less than n_{serv} non-passive sessions against the server oracle, n_{term} non-passive sessions against the terminal oracle, $n_{\mathsf{total}} > n_{\mathsf{term}} + n_{\mathsf{serv}}$ total sessions and $T < n_{\mathsf{total}}$ unique timeframes. Then there exist an adversary $\mathcal{D}_1$ against the indistinguishability of the password-distribution $\mathcal{D}$ running in time t, an adversary $\mathcal{D}_5$ against the DDH experiment running in time $t + 8n_{\mathsf{total}}\tau_{exp}$, four adversaries $\mathcal{B}'_3$, $\mathcal{B}'_5$, $\mathcal{D}_2$, and $\mathcal{D}_3$ against the commitment scheme properties running in time t:*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \times 2^{-D} + \mathsf{Adv}^{\mathsf{dist}\text{-}T}_{F}(\mathcal{D}_1) + \mathsf{Adv}^{\mathsf{ind}}_{\mathit{DDH}}(\mathcal{D}_5) + \mathsf{Adv}^{\mathsf{binding}}_{\mathcal{CS}}(\mathcal{B}'_3)\\ + \mathsf{Adv}^{\mathsf{setup}\text{-}\mathsf{ind}}_{\mathcal{CS}}(\mathcal{D}_2) + n_{\mathsf{total}} \times \mathsf{Adv}^{\mathsf{s}\text{-}\mathsf{eq}}_{\mathcal{CS}}(\mathcal{D}_3) + \mathsf{Adv}^{\mathsf{s}\text{-}\mathsf{binding}}_{\mathcal{CS}}(\mathcal{B}'_5)\\ \mathsf{Adv}^{\mathsf{u}\text{-}\mathsf{auth}}_{\mathsf{HAKE}}(\mathcal{A}') \leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \times 2^{-D} + \mathsf{Adv}^{\mathsf{dist}\text{-}T}_{F}(\mathcal{D}_1) + \mathsf{Adv}^{\mathsf{setup}\text{-}\mathsf{ind}}_{\mathcal{CS}}(\mathcal{D}_2)\\ + n_{\mathsf{total}} \times \mathsf{Adv}^{\mathsf{s}\text{-}\mathsf{eq}}_{\mathcal{CS}}(\mathcal{D}_3) + \mathsf{Adv}^{\mathsf{s}\text{-}\mathsf{binding}}_{\mathcal{CS}}(\mathcal{B}'_5),\end{aligned}$$

with τ_{exp} the time necessary to exponentiate one group element, and n_{total} the global number of sessions.

Proof. Once again, we will denote $\mathcal{B}_i$ the simulator for game $\mathbf{G}_i$, that outputs 1 if $b \leftarrow \mathcal{A}$ and 0 otherwise and define $\mathcal{D}_i$ the distinguisher between games $\mathbf{G}_i$ and $\mathbf{G}_{i-1}$, for $i > 1$.

Game $\mathbf{G}_0$: This is the real game, where the adversary outputs its guess on the bit b:

$$\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) = 2 \times \Pr_{\mathbf{G}_0}[b \leftarrow \mathcal{A}] - 1.$$

Game $\mathbf{G}_1$: In this game the simulator will execute the real protocol, generating a password $\mathsf{pw}_t \xleftarrow{\$} \mathcal{D}$ at the beginning of each timeframe t and subsequently using it whenever necessary. We will also consider a flag, `NOG-Com-OK`, that can be raised during the execution of the simulation. More precisely, in $\mathbf{G}_1$, the simulator answers each request as follows:

1. `SendServ`$(k, (\ell, X_T, c_T))$: Set the current timeframe t to be π^k_S's session timeframe. Generate $x_S \xleftarrow{\$} \mathbb{Z}_p$ and $X_S \leftarrow g^{x_S}$. Then set $(c_S, s_S) \leftarrow \mathsf{Com}(X_S, \mathsf{pw}_t)$ and send (X_S, c_S).
2. `SendServ`(k, s_T): Check whether $\mathsf{Open}(c_T, s_T) = (X_T, \mathsf{pw}_t)$.
   - If so, send s_S, accept and set $\mathsf{sk}_S = (X_T)^{x_S}$. Additionaly, if π^k_S is *fresh* and c_T was not *oracle-generated*, raise flag `NOG-Com-OK`.
   - If the equality is not verified reject.
3. `SendTerm`$(j, $ `Start`$)$: Set the current timeframe t to be π^j_T's session timeframe. Generate $x_T \xleftarrow{\$} \mathbb{Z}_p$ and $X_T \leftarrow g^{x_T}$. accept on behalf of the human and use the current password pw_t to set $(c_T, s_T) \leftarrow \mathsf{Com}(X_T, \mathsf{pw}_t)$ and send (ℓ, X_T, c_T). Additionally, if π^j_T is *compromised*, reveal the password pw_t.

4. $\texttt{SendTerm}(j, (X_S, c_S))$: Wait until the timeframe is $> t$, then send s_T.
5. $\texttt{SendTerm}(j, s_S)$: Check whether $\mathsf{Open}(c_S, s_S) = (X_S, \mathsf{pw}_t)$.
   - If so, set $\mathsf{sk}_T = (X_S)^{x_T}$. Additionaly, if π_T^j is *fresh* and c_S was not *oracle-generated*, raise flag NOG-Com-OK.
   - If the equality is not verified, reject.
6. $\texttt{SendHum}()$: If π_T^j was marked as *infected*, reveal pw_t, where t is the *current* timeframe and accept.
7. $\texttt{Compromise}(j)$: Mark π_T^j as *compromised* and reveal the random tape.
8. $\texttt{Infect}(j)$: If π_T^j is *compromised* through a previous $\texttt{Compromise}(j)$ query, mark it as *infected*, allowing SendHum queries.
9. $\texttt{Test}(j, P)$: according to b and whether π_P^j is fresh or not, the real key sk_P, a random key or $\perp$ is returned.

It should be clear that this simulation performs exactly as the real game should, except for using the global distribution $\mathcal{D}$ to generate the passwords, since the additional flags are purement formal but do not affect the simulation. Hence:

$$|\Pr_{\mathbf{G}_1}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_0}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}_F^{\mathsf{dist}\text{-}T}(\mathcal{D}_1),$$

where $\mathcal{D}_1$ behaves as $\mathcal{B}_0$ but using either distribution $\mathcal{D}$ (which is then $\mathbf{G}_1$) or the real distribution (as in $\mathbf{G}_0$) to generate the passwords pw_t.

Game $\mathbf{G}_2$: In this game, unless the oracle is *compromised*, we will reject all openings for *non-oracle-generated* commitments.

More precisely, we change the queries answers as follows:

2. $\texttt{SendServ}(k, s_T)$: If c_T was *oracle-generated* and $\mathsf{Open}(c_T, s_T) = (X_T, \mathsf{pw}_t)$, send s_S, accept and set $\mathsf{sk}_S(X_S)^{x_T}$. Otherwise, reject.
5. $\texttt{SendTerm}(j, s_S)$:
   - If *compromised*, act exactly as in $\mathbf{G}_1$
   - Otherwise: If c_S was *oracle-generated* and $\mathsf{Open}(c_S, s_S) = (X_S, \mathsf{pw}_t)$, accept and set $\mathsf{sk}_T = (X_S)^{x_T}$. Otherwise, reject.

Obviously, games $\mathbf{G}_2$ and $\mathbf{G}_1$ are not indistinguishable. However, this can only make a difference when NOG-Com-OK was raised. Hence:

$$|\Pr_{\mathbf{G}_2}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_1}[b \leftarrow \mathcal{A}]| \leq \Pr_{\mathbf{G}_1}[\texttt{NOG-Com-OK}]$$

Game $\mathbf{G}_3$: In this game, we further straighten our requirements for openings in *non-compromised* terminals. Indeed, we will also reject an opening if it opens to a different X_S/X_T than the one it was initially generated for. More precisely:

1. $\texttt{SendServ}(k, (\ell, X_T, c_T))$: Act as in $\mathbf{G}_1$. Then store $(c_S, X_S, s_S) \in \Upsilon_S$.
2. $\texttt{SendServ}(k, s_T)$: If $\mathsf{Open}(c_T, s_T) = (X_T, \mathsf{pw}_t)$ and $(c_T, X_T, \cdot) \in \Upsilon_T$, send s_S, accept and set $\mathsf{sk}_S = (X_S)^{x_T}$. Otherwise, reject.
3. $\texttt{SendTerm}(j, \texttt{Start})$: Act as in $\mathbf{G}_1$. Then store $(c_T, X_T, s_T) \in \Upsilon_T$.
5. $\texttt{SendTerm}(j, s_S)$:
   - If *compromised*, act exactly as in $\mathbf{G}_2$
   - Otherwise: If $\mathsf{Open}(c_S, s_S) = (X_S, \mathsf{pw}_t)$ and $(c_S, X_S, \cdot) \in \Upsilon_S$, set $\mathsf{sk}_T = (X_S)^{x_T}$. Otherwise, reject.

Game $\mathbf{G}_3$ and $\mathbf{G}_2$ are almost the same. The only difference would be if $\mathcal{A}$ somehow reuses an *oracle-generated* commitment c_P but opens it to a different key X_P^* (and the same, valid, password pw_t) than the *oracle-generated* one X_P. This would, however, break the *binding* property of $\mathcal{CS}$, as we now know two opening values (s_P, s_P^*) for two messages $((X_P, \mathsf{pw}_t), (X_P^*, \mathsf{pw}_t))$ with the same commitment c_P; we define a simulator $\mathcal{B}_3'$ that behaves as $\mathcal{B}_3$ but outputs such a tuple.

As it happens only on *non-passive* sessions, of which there is at most $n_{\mathsf{term}} + n_{\mathsf{serv}}$, we have:

$$|\Pr_{\mathbf{G}_3}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_2}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{binding}}(\mathcal{B}_3').$$

Game $\mathbf{G}_4$: We now consider a CDH tuple $(A, B, C) = (g^a, g^b, g^{ab})$ and embed it into the simulation. Once again, simulations still behave as in the original game $\mathbf{G}_1$ for *compromised* sessions. More precisely, we change the queries answers from game $\mathbf{G}_3$ as follows:

1. $\texttt{SendServ}(k, (\ell, X_T, c_T))$: Set the current timeframe t to be π_S^k's session timeframe. Generate $(\beta, \delta) \xleftarrow{\$} \mathbb{Z}_p^2$ and set $X_S \leftarrow B^\delta \cdot g^\beta$. Then set $(c_S, s_S) \leftarrow \mathsf{Com}(X_S, \mathsf{pw}_t)$ and send (X_S, c_S). Lastly, store $(c_S, X_S, s_S) \in \Upsilon_S$.

3. $\texttt{SendTerm}(j, \texttt{Start})$:
   - If *compromised*, act exactly as in $\mathbf{G}_3$.
   - Otherwise: Set the current timeframe t to be π_T^j's session timeframe. Generate $(\alpha, \gamma) \xleftarrow{\$} \mathbb{Z}_p^2$ and set $X_T \leftarrow A^\gamma \cdot g^\alpha$. Use the current password pw_t to set $(c_T, s_T) \leftarrow \mathsf{Com}(X_T, \mathsf{pw}_t)$ and send (ℓ, X_T, c_T). Then store $(c_T, X_T, s_T) \in \Upsilon_T$.

9. $\texttt{Test}(j, P)$: If sk_P has been generated and π_P^j is *fresh*, we know $(\alpha, \beta, \gamma, \delta)$ that were used to construct X_T and X_S (we only compute sk_P if both are *oracle-generated*). Then, the simulator uses $\mathsf{sk}_P = C^{\gamma\delta} \cdot B^{\alpha\delta} \cdot A^{\beta\gamma} \cdot g^{\alpha\beta}$ for the real key. Otherwise, it returns $\perp$.

Regarding *compromised* sessions, either the adversary learned the password pw (from $\texttt{SendHum}$-query or by letting the *compromised* terminal ask the user) and no $\texttt{Test}$-query can be asked as neither π_T^j nor π_S^k is fresh; or the adversary did not learn the password, in which case $\texttt{Test}$-queries can still be asked to π_S^k, but the compromise did not reveal any secret.

Since we have $C = g^{ab}$, then $\mathsf{sk}_P = g^{(a\gamma+\alpha)\cdot(b\delta+\beta)}$ is exactly as the real key should be. This game is perfectly indistinguishable from the previous one:

$$\Pr_{\mathbf{G}_4}[b \leftarrow \mathcal{A}] = \Pr_{\mathbf{G}_3}[b \leftarrow \mathcal{A}].$$

Game $\mathbf{G}_5$: We are now given a random C, independent of A and B:

Hence, using a distinguisher $\mathcal{D}_5$, that behaves as $\mathcal{B}_5$ with C either real (as in $\mathbf{G}_4$) or random (which is $\mathbf{G}_5$):

$$|\Pr_{\mathbf{G}_5}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_4}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{DDH}}(\mathcal{D}_5),$$

Note that our simulator makes 8 exponentiations to simulate any session in addition to the running time of $\mathcal{A}$, so $\mathcal{D}_5$ runs in time $t_\mathcal{A} + 8n_{\mathsf{total}}\tau_{\text{exp}}$ with $t_\mathcal{A}$ the running time of $\mathcal{A}$ and τ_{exp} the time necessary to exponentiate one group element.

When C is random, then every sk_P is random and independent of all others. Hence one cannot distinguish the real key from a random key, since they are both random: $\Pr_{\mathbf{G}_5}[b \leftarrow \mathcal{A}] = 1/2$.

As a consequence,

$$\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{HAKE}}(\mathcal{A}) \leq \Pr_{\mathbf{G}_1}[\texttt{NOG-Com-OK}] + \mathsf{Adv}^{\mathsf{dist}\text{-}T}_{F}(\mathcal{D}_1) + \mathsf{Adv}^{\mathsf{ind}}_{\mathsf{DDH}}(\mathcal{D}_5) + \mathsf{Adv}^{\mathsf{binding}}_{\mathcal{CS}}(\mathcal{B}'_3).$$

We now need to bound the probabilities of events $\texttt{NOG-Com-OK}$ in game $\mathbf{G}_1$. For this, we construct another series of games from game $\mathbf{G}_1$, in which we will consider the probability of raising a flag instead of the regular advantage over the $\texttt{Test}$-queries. For the sake of simplicity, we will denote the simulators in this branch $\mathcal{C}_i$ for game $\mathbf{G}_i$.

Game $\mathbf{G}_2$: In this game, we go back to game $\mathbf{G}_1$ and change the simulator's behavior to make use of the equivocality property of our commitment scheme. But before any equivocation, we just change the setup procedure of the commitment:

$$|\Pr_{\mathbf{G}_2}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_1}[b \leftarrow \mathcal{A}]| \leq \mathsf{Adv}^{\mathsf{setup\text{-}ind}}_{\mathcal{CS}}(\mathcal{D}_2),$$

where $\mathcal{D}_2$ is a distinguisher that behaves as $\mathcal{B}_1$, with either Setup (which is $\mathbf{G}_1$) or Setup' (which is $\mathbf{G}_2$).

Game $\mathbf{G}_3$: The simulator can now use its oracle access to equivocal commitments whenever it has to use commitments in order to delay the actual committed value. However, if the terminal is *compromised*, we will still use regular commitments algorithm. This is due to the fact that the adversary knows the random tape and can therefore deterministically reproduce the output of Com. The simulator runs as follows:

1. $\texttt{SendServ}(k, (\ell, X_T, c_T))$: Set the current timeframe t to be π_S^k's session timeframe. Generate $x_S \xleftarrow{\$} \mathbb{Z}_p$ and $X_S \leftarrow g^{x_S}$. Get $c_S \leftarrow \mathsf{GenEquivCommit}$ and send (X_S, c_S).
2. $\texttt{SendServ}(k, s_T)$: Check whether $\mathsf{Open}(c_T, s_T) = (X_T, \mathsf{pw}_t)$.
   - If so, send $s_S \leftarrow \mathsf{OpenEquivCommit}(c_S, (X_S, \mathsf{pw}_t))$, accept and output $\mathsf{sk}_S = (X_T)^{x_S}$. Additionally, if π_S^k is *fresh* and c_T was not *oracle-generated*, raise flag NOG-Com-OK.
   - Otherwise output $\perp$.
3. $\texttt{SendTerm}(j, \texttt{Start})$: Set the current timeframe t to be π_T^j's session timeframe and generate $x_T \xleftarrow{\$} \mathbb{Z}_p$ and $X_T \leftarrow g^{x_T}$. Then:
   - If *compromised*: Act as in $\mathbf{G}_1$.
   - If *uncompromised*: Get $c_T \leftarrow \mathsf{GenEquivCommit}$ and send (ℓ, X_T, c_T).
4. $\texttt{SendTerm}(j, (X_S, c_S))$: Wait until the timeframe is $> t$. Then:
   - If *compromised*: Act as in $\mathbf{G}_1$.
   - If *uncompromised*: Send $s_T \leftarrow \mathsf{OpenEquivCommit}(c_T, (X_T, \mathsf{pw}_t))$.
5. $\texttt{SendTerm}(j, s_S)$: Check whether $\mathsf{Open}(c_S, s_S) = (X_S, \mathsf{pw}_t)$.
   - If so, generate and output $\mathsf{sk}_T = (X_S)^{x_T}$ Additionally, if π_T^j is *fresh* and c_S was not *oracle-generated*, raise flag NOG-Com-OK.
   - Otherwise output $\perp$.

The difference is just in the use of equivocal commitments instead of real ones:

$$|\Pr_{\mathbf{G}_3}[b \leftarrow \mathcal{A}] - \Pr_{\mathbf{G}_2}[b \leftarrow \mathcal{A}]| \leq n_{\mathsf{total}} \times \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{s\text{-}eq}}(\mathcal{D}_3).$$

where $\mathcal{D}_3$ is a distinguisher that behaves as $\mathcal{C}_2$ but uses either real (like in $\mathbf{G}_2$) or equivocal (as in $\mathbf{G}_3$) commitments.

Game $\mathbf{G}_4$: One could remark that in the previous game, unless the session is *compromised*, the simulator doesn't use pw_t in any timeframe $\leq t$. Therefore, in this game, the simulator will delay producing pw_t until the timeframe t has ended, unless Compromise is called. This can be done using the same oracle handlers as in $\mathbf{G}_3$, simply adding the password definition:

2. $\texttt{SendServ}(k, s_T)$: If not yet generated, generate $\mathsf{pw}_t \xleftarrow{\$} \mathcal{D}$. Then act as in $\mathbf{G}_3$.
4. $\texttt{SendTerm}(j, (X_S, c_S))$: After waiting for the current timeframe to be $> t$, if not yet generated, generate $\mathsf{pw}_t \xleftarrow{\$} \mathcal{D}$. Then act as in $\mathbf{G}_3$.
7. $\texttt{Compromise}(j)$: Generate $\mathsf{pw}_t \xleftarrow{\$} \mathcal{D}$. Then act as in $\mathbf{G}_3$.

This game is perfectly indistinguishable from the previous one:

$$\Pr_{\mathbf{G}_4}[b \leftarrow \mathcal{A}] = \Pr_{\mathbf{G}_3}[b \leftarrow \mathcal{A}].$$

Game $\mathbf{G}_5$: We will now use the extractability property of our commitment scheme to check the adversary's commitment, by modifying the simulator as follows:

1. $\texttt{SendServ}(k, (\ell, X_T, c_T))$: If c_T was not *oracle-generated*, extract $(X_T^*, \mathsf{pw}_t^*) \leftarrow \mathsf{ExtractCommit}(c_T)$. Then act as in $\mathbf{G}_4$.
2. $\texttt{SendServ}(k, s_T)$: If c_T was *oracle-generated*, act as in $\mathbf{G}_4$. Otherwise, generate $\mathsf{pw}_t \xleftarrow{\$} \mathcal{D}_t$. Then check whether $(X_T^*, \mathsf{pw}_t^*) = (X_T, \mathsf{pw}_t)$.
   - If so, send $s_S \leftarrow \mathsf{OpenEquivCommit}(c_S, (X_S, \mathsf{pw}_t))$, accept and output $\mathsf{sk}_S = (X_T)^{x_S}$. Additionally, if π_S^k is *fresh*, raise flag NOG-Com-OK.
   - Otherwise output $\perp$.
4. $\texttt{SendTerm}(j, (X_S, c_S))$: Extract $(X_S^*, \mathsf{pw}_t^*) \leftarrow \mathsf{ExtractCommit}(c_S)$. Then act as in $\mathbf{G}_4$.
5. $\texttt{SendTerm}(j, s_S)$: If c_S was *oracle-generated*, act as in $\mathbf{G}_4$. Otherwise, check whether $(X_S^*, \mathsf{pw}_t^*) = (X_S, \mathsf{pw}_t)$.
   - If so, generate and output $\mathsf{sk}_T = (X_S)^{x_T}$ Additionally, if π_T^j is *fresh* and c_S was not *oracle-generated*, raise flag NOG-Com-OK.
   - Otherwise output $\perp$.

Note that in this game, except for compromised sessions, pw_t^* is obtained before pw_t is generated. Hence $\Pr[\mathsf{pw}_t = \mathsf{pw}_t^*] = 2^{-D}$.
The only way to distinguish Game $\mathbf{G}_5$ from Game $\mathbf{G}_4$ would be if $(\mathsf{Open}(c_P, s_P) \neq \mathsf{ExtractCommit}(c_P)$. But if so, then (c_P, s_P) breaks the *strong binding extractability*, which can be outputed by a simulator $\mathcal{B}_5'$ similar to $\mathcal{C}_5$. Hence:

$$|\Pr_{\mathbf{G}_5}[\texttt{NOG-Com-OK}] - \Pr_{\mathbf{G}_4}[\texttt{NOG-Com-OK}]| \leq \mathsf{Adv}_{\mathcal{CS}}^{\text{s-binding}}(\mathcal{B}_5').$$

Moreover, we can now bound the probability that those flags are raised in game $\mathbf{G}_5$. Indeed, for it to happen, there must exist a oracle π_P^j such that $(X_P^*, \mathsf{pw}_t^*) = (X_P, \mathsf{pw}_t)$. In particular, we must have $\mathsf{pw}_t^* = \mathsf{pw}_t$. Moreover, this cannot happen if the session was *passive*. Hence,

$$\Pr_{\mathbf{G}_5}[\texttt{NOG-Com-OK}] \leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \times 2^{-D}.$$

And so:

$$\Pr_{\mathbf{G}_1}[\texttt{NOG-Com-OK}] \leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \times 2^{-D} + \mathsf{Adv}_{\mathcal{CS}}^{\text{setup-ind}}(\mathcal{D}_2) + n_{\mathsf{total}} \times \mathsf{Adv}_{\mathcal{CS}}^{\text{s-eq}}(\mathcal{D}_3) + \mathsf{Adv}_{\mathcal{CS}}^{\text{s-binding}}(\mathcal{B}_5').$$

In conclusion, we have various adversaries such that:

$$\begin{aligned}\mathsf{Adv}_{\mathsf{HAKE}}^{\mathsf{priv}}(\mathcal{A}) \leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \times 2^{-D} + \mathsf{Adv}_{F}^{\text{dist-}T}(\mathcal{D}_1) + \mathsf{Adv}_{\mathsf{DDH}}^{\mathsf{ind}}(\mathcal{D}_5) + \mathsf{Adv}_{\mathcal{CS}}^{\mathsf{binding}}(\mathcal{B}_3') \\ + \mathsf{Adv}_{\mathcal{CS}}^{\text{setup-ind}}(\mathcal{D}_2) + n_{\mathsf{total}} \times \mathsf{Adv}_{\mathcal{CS}}^{\text{s-eq}}(\mathcal{D}_3) + \mathsf{Adv}_{\mathcal{CS}}^{\text{s-binding}}(\mathcal{B}_5').\end{aligned}$$

Proof (Authenticity proof). The simulated game $\mathbf{G}_1$ is indistinguishable from the real world (be it in the privacy security game or the authentication security game).
To break user authentication means that the adversary successfully opened a commitment c_T to X_T, pw_t without interacting with U_ℓ. This is exactly the situation in which flag `NOG-Com-OK` is raised. Hence:

$$\mathsf{Adv}_{\mathsf{HAKE}}^{\text{u-auth}}(\mathcal{A}') \leq \Pr_{\mathbf{G}_1}[\texttt{NOG-Com-OK}] + \mathsf{Adv}_{F}^{\text{dist-}T}(\mathcal{D}_1)$$

Using the same results about $\Pr_{\mathbf{G}_1}[\texttt{NOG-Com-OK}]$, this gives:

$$\begin{aligned}\mathsf{Adv}_{\mathsf{HAKE}}^{\text{u-auth}}(\mathcal{A}') \leq (n_{\mathsf{serv}} + n_{\mathsf{term}}) \times 2^{-D} + \mathsf{Adv}_{\mathcal{CS}}^{\text{setup-ind}}(\mathcal{D}_2) + n_{\mathsf{total}} \times \mathsf{Adv}_{\mathcal{CS}}^{\text{s-eq}}(\mathcal{D}_3) \\ + \mathsf{Adv}_{\mathcal{CS}}^{\text{s-binding}}(\mathcal{B}_5') + \mathsf{Adv}_{F}^{\text{dist-}T}(\mathcal{D}_1)\end{aligned}$$