

# HPAKE: Honey Password-authenticated Key Exchange for Fast and Safer Online Authentication

Wenting Li<sup>1</sup>, Ping Wang<sup>2</sup>, Senior Member, IEEE, Kaitai Liang<sup>3</sup>, Member, IEEE,

<sup>1</sup>School of Computer Science, Peking University, Beijing 100871, China (wentingli@pku.edu.cn).

<sup>2</sup>National Engineering Research Center for Software Engineering, Beijing 100871, China (pwang@pku.edu.cn)

<sup>2</sup>School of Software & Microelectronics, Peking University, Beijing 100871, China

<sup>2</sup>Key Laboratory of High Confidence Software Technologies, Ministry of Education(Peking University), China

<sup>3</sup>Department of Intelligent Systems, Delft University of Technology, 2628 Delft, The Netherlands (kaitai.liang@tudelft.nl)

Password-only authentication is one of the most popular secure mechanisms for real-world online applications. But it easily suffers from a practical threat - password leakage, incurred by external and internal attackers. The external attacker may compromise the password file stored on the authentication server, and the insider may deliberately steal the passwords or inadvertently leak the passwords. So far, there are two main techniques to address the leakage: Augmented password-authentication key exchange (aPAKE) against insiders and honeyword technique for external attackers. But none of them can resist both attacks. To fill the gap, we propose the notion of *honey PAKE (HPAKE)* that allows the authentication server to detect the password leakage and achieve the security beyond the traditional bound of aPAKE. Further, we build an HPAKE construction on the top of the honeyword mechanism, honey encryption, and OPAQUE which is a standardized aPAKE. We formally analyze the security of our design, achieving the insider resistance and the password breach detection. We implement our design and deploy it in the real environment. The experimental results show that our protocol only costs 71.27 ms for one complete run, within 20.67 ms on computation and 50.6 ms on communication. This means our design is secure and practical for real-world applications.

**Index Terms**—Password, honeyword, leakage detection, password-authenticated key exchange.

## I. INTRODUCTION

**P**ASSWORD-ONLY AUTHENTICATION (PoA), which has great advantages on usability and deployability, is one of the most popular online authentication methods [1]. It has been attracting attentions from academia, and recently many research works have been proposed in this field [2], [3], [4]. But PoA does easily suffer from password leakage. Billions of personal and business passwords have been compromised by hackers [5] which yields a considerable amount of users' privacy leakage and financial loss, for example, Yahoo made 3 billion accounts exposed [6] and it finally agreed to settle for 117.5 million US dollars [7]. The password leakage, in practice, may be caused by: 1) active external attacks (e.g., SQL injection), or 2) the internal design flaws and software bugs (for instance, GitHub records passwords in plaintext [8]). It is not trivial to handle these attacks in the context of PoA.

### A. Existing Solutions

*Against Insiders.* In Figure 1a, Augmented password-authentication key exchange (aPAKE) [9] is designed to allow a client and a server to establish a session key based on a password, where the client has the password plaintext and the server only holds the verifier. This technique prevents the server from knowing the password, and therefore resists the

insider attacks. Since Bellovin and Merrit [9] introduced this notion, many researchers proposed various aPAKE schemes [10], [11], [12], [13] in order to improve the security and efficiency performance. Among them, OPAQUE [12] is the most well-studied scheme with the strongest security and thus, it recently is standardized by the Crypto Forum Research Group of the Internet Engineering Task Force (IETF) [14].

*Against Outsiders.* Honeyword technique [15] (see Figure 1b) is proposed to detect the password leakage for the most common password-only authentication systems, password-over-TLS. This approach associates  $t-1$  decoy and plausible-looking passwords (i.e., honeywords) to each account. The honeywords and the real password are collectively called sweetwords. If an attacker steals the password file, she cannot tell the real one and probably (with  $1-1/t$  probability) log in with a honeyword. Then, the server can detect the password leakage from the “wrong” login. The follow-up works focus on the honeyword generation algorithms [16], [17] so as to produce more plausible-looking decoys and the detection methods [18] to improve reliability.

*Others.* Passwordless authentication [19] or multi-factor authentication systems [20], [21] make good use of other factors, e.g., smartphone and fingerprint. They significantly reduce the risk of password leakage. If an attacker steals the password, she still needs additional factors to compromise account. Besides, in some of these designs, authentication server does not need to store the password-related data, so that even if the attacker compromises the storage file on server, she cannot carry out offline password guessing as long as other factors are secure. A typical design can be seen in [21], [22], [23] that a smart device (as an authentication factor) is used to

This research is supported by National Key R&D Program of China (2020YFB1805400), National Natural Science Foundation of China (62072010), China Postdoctoral Science Foundation (2021M700215), European Union's Horizon 2020 research and innovation programme under grant agreement No. 952697 (ASSURED), No. 101021727 (IRIS), and No. 101070052 (TANGO), and High-performance Computing Platform of Peking University. (Corresponding author: Ping Wang).

store the password-related data, making systems resist offline guessing in the case of server compromise.

*Shortcomings.* The techniques above, unfortunately, have the following shortcomings. The honeyword mechanism requires the client to send the password plaintext to the server (via a server-authenticated secure channel), otherwise the server cannot tell if the login password is real. Thus an insider can directly steal the plaintext of the login password without any guessing attacks. In aPAKE, the server has to store the verifiers in the password file for authentication. But an external attacker may steal the file and carry out guessing attacks [24] to recover the password. This vulnerability is inherent in aPAKE. And neither of these methods can provide a solution maintaining security against both insiders and outsiders. As for other (passwordless or multi-factor) approaches, they may provide stronger security relying on extra factors, which may bring disadvantages to deployability and usability. In this paper, we do not consider them and only focus on password-only authentication. According to the above discussion, we thus raise a question: “How could one design a fast and secure password-only authentication scheme that can resist both the insider and external attackers?”

### B. A New Solution: Honey PAKE

To answer the question, we propose the notion of honey PAKE (HPAKE) by combining the aPAKE and honeyword techniques. This combination is shown in Figure 1c, in which HPAKE sets up  $t - 1$  honeywords for each account and stores their verifiers on the authentication server. From Table I, we see that HPAKE inherits the advantages rather than the shortcomings of the aPAKE and honeyword techniques. Specifically:

- 1) For the external attacker: HPAKE provides the password leakage as well as honeyword techniques. If the authentication server is compromised, the attacker will get a password list including  $t$  sweetwords via offline guessing attacks. The attacker cannot tell which one is real and probably runs HPAKE with a honeyword to compromise the account. This will produce a honey session key, and the usage of the session key will be detected and alarmed.
- 2) For the insider: HPAKE guarantees that the password plaintext is never left from the client, achieving the same security as aPAKE. The authentication is explicitly done by the key exchange. And running with the real password will yield a real session key, and only the instruction encrypted by the real session key will be allowed. Therefore, the server/insider cannot steal the password plaintext.

**Design challenge.** The idea of HPAKE is natural, but how to design a concrete and secure HPAKE is full of challenges. The first one is that the ways of handling passwords between honeyword mechanism and aPAKE are different. The password plaintext has to be sent to server for verification for the former, while the latter does not allow the password to leave the client. One may come up with a trivial solution here: running  $t$  aPAKE instances in parallel. Specifically, the

authentication server executes  $t$  aPAKE instances, and each of them uses one sweetword; the client also run  $t$  instances, but all of them use its (real) password. For an aPAKE with explicit authentication, only the instance where the client password is equal to the server sweetword will yield a session key. But this approach increases the computational and communication cost by the number of sweetwords. Furthermore, it decreases the resistance against online password guessing by a factor of  $t$ : an attacker (without the password file) can run an HPAKE instance to verify the correctness of  $t$  password guesses (i.e. running the  $t$  aPAKE instances with the  $t$  password guesses, as the server, to interact with the client).

### C. Our Contribution

**A novel design for HPAKE.** Right after defining the notion of HPAKE, we propose a novel construction which is based on OPAQUE [12] and honeyword mechanism, and introduce honey encryption to replace the encryption scheme used in OPAQUE. Specifically, on the client side, our HPAKE keeps the same user interface as OPAQUE. Since OPAQUE is standardized by IETF [14], our construction achieves the advantages on deployability, which might be of independent interest. On the server side: 1) at the registration phase, our HPAKE generates honeywords, runs the same registration process for them as the real password to generate the user’s honey private/public keys, and stores the honey public keys along with the real one on the server; 2) at the authentication, the server runs the key-exchange process, calculates several session keys with different user’s public keys, and checks if the real session key or a honey one is used by the client to detect the password leakage. The key of our design is to use honey encryption [25] to generate the user’s honey private keys by decrypting the ciphertext of the real user’s private key for honeywords. Therefore, our HPAKE does not need to run several OPAQUE instances, but just calculates several session keys on the server side by sharing the same ciphertext for the real password and honeywords.

**Security analysis.** To formally analyze the security of our design, we propose a game-based security model that captures the security of both aPAKE and honeyword mechanism. For the former, our model achieves the security against the insider attacks, which are considered in the traditional aPAKE security model, and against the online/offline password guessing attacks. In the latter, we capture the ability of detecting password leakage by defining the indistinguishability between the real and honey session keys. We further formally prove the security of our design under the specified model. This means our design can provide stronger security than both aPAKE and honeyword mechanism. Specifically, if the authentication server is compromised, HPAKE can detect the password leakage which is safer than OPAQUE; otherwise, HPAKE is as safe as OPAQUE against password guessing attacks and insider attacks which is safer than honeyword mechanism for password-over-TLS.

**Implementation and efficiency analysis.** Compare with OPAQUE, our design slightly increases the cost on computation and obtains the same communication complexity between client and server. Client takes 3 exponentiations and 2 multi-

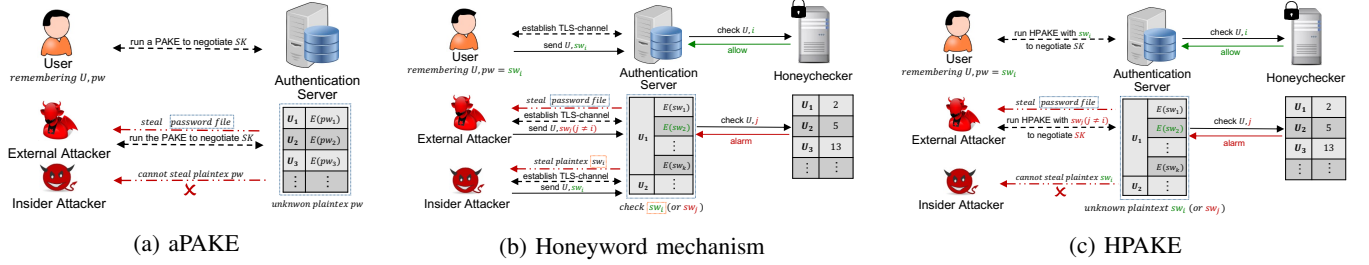


Fig. 1: Overview on aPAKE, honeyword mechanism, and HPAKE

TABLE I: Our HPAKE and the existing techniques

| Technique              | Literature       | Design   | Advantage   | Disadvantage  |
|------------------------|------------------|--|---|---|
| aPAKE                  | [9], [12]        | Cryptography protocols built on OPRF, DH key exchange, etc.            | Prevent the server/insider from knowing the password plaintext. | Suffer from offline password guessing if the password file is leaked. |
| Honeyword <sup>1</sup> | [15], [18], [17] | Storing decoy passwords along with the real password for each account. | Detect the password leakage if the password file is stolen.     | Allow the server/insider to know the password plaintext.              |
| Our HPAKE              | -                | Combining the above two techniques (see Section III).                  | Both of the above two.  | None of the above two.  |

<sup>1</sup> Honeyword mechanism is only designed for password-over-TLS.

exponentiations, while server costs 3 exponentiations and  $t$  multi-exponentiations, where  $t$  is the number of sweetwords for each account. We note that the multi-exponentiation is linear with the number of honeyword. We further implement and deploy the design in the real-world environment for efficiency evaluation. The experimental results show that our design only costs 3.48 ms and 17.19 ms on the client and server sides (by setting  $t = 20$ ), and 50.48 ms for the communication between them. This means our HPAKE is fast and efficient for real-world online authentication.

## II. PRELIMINARY AND RELATED WORKS

### A. Honeyword Mechanism

Honeyword technique [15] has been proposed to detect password leakage for password-over-TLS. As shown in Figure 1b, honeyword mechanism directly generates several honeywords and stores them (in the form of hash value) on the authentication server along with the real password. It stores the index of the real password in the list on another server called honeychecker. When one logs in with a username  $U$  and a password  $pw$  (where  $pw$  is sent to the authentication server via the TLS channel), then the authentication server checks if  $pw$  is a sweetword:

- 1) If it is not, deny this login.
- 2) If it is the  $i$ -th sweetword, the authentication server sends  $(U, i)$  to the honeychecker (via a secure channel). Then the honeychecker checks if the index  $i$  is correct for  $U$ :
  - a) If it is, allow this login.
  - b) Otherwise, raise an alarm of password leakage and take actions according to the pre-defined security policy.

This mechanism only does slight modification on the server side for password-over-TLS, and therefore maintains its advantages on deployability. Besides, since its interface is very simple, the honeychecker can be easily enhanced to avoid being compromised.

### B. OPAQUE

OPAQUE is a strong aPAKE proposed in 2018 [12] and is standardized by IETF [26] recently. Password-authenticated key exchange (PAKE) [27], [12] allows two parties sharing a low-entropy password to establish an authenticated session key in the basic case where the two parties do not rely extra mechanisms (e.g., PKI) except the communication with each other. PAKE can prevent the eavesdropper from getting the password even if she carries out offline password guessing attacks. There are two types of PAKE: symmetric PAKE (also known as balanced PAKE) [27] and asymmetric PAKE (also called augmented PAKE) [9], [12]. The former considers the symmetric case where both of the parties (two humans) hold the plaintext of password, and the latter focuses on the asymmetric (client-server) case where one party (the client) holds the password plaintext, and the other one (the server) stores a verifier of the password (e.g., the hash values of the password). In the asymmetric setting, aPAKE prevents the server from knowing the password plaintext and forces the attacker to carry out offline password guessing if she wants to impersonate the user from the compromised password file (the verifier of the password stored on the server). OPAQUE, as a state-of-the-art aPAKE, further provides stronger security, i.e., resisting pre-computing attacks.

In OPAQUE, the client  $C$  (as the user  $U$ ): 1) runs OPRF (oblivious pseudorandom function) with the server  $S$  to convert the low-entropy password  $pw$  to a high-entropy secret  $rw$ , called random password, 2) uses  $rw$  to decrypt the ciphertext  $c$  obtained from the server to yield the user's private key  $k_U$  and the server's public key  $K_S$ , and 3) finally runs AKE (authenticated key exchange) for key exchange with  $k_U$  and  $K_S$ . In [12], the instantiated OPAQUE uses 2HashDH [28] and HMQV [29] as OPRF and AKE, respectively.

Note any PAKE protocols cannot prevent the attacker from impersonating the server to carry out online password guessing attacks, since the authentication only relies on the password.

### Parameter

- Security parameter  $\kappa$ .
- 2HashDH:
  - 1)  $m_1$  is a primer number,  $G_1$  is a  $m_1$ -order cyclic group, and  $g_1$  is a generator of  $G_1$ . The length of  $m_1$  is a polynomial function of  $\kappa$ .
  - 2)  $H_1, H'_1$  are two hash functions with ranges  $\{0, 1\}^{l_1}$  and  $G_1$ , respectively. The PRF  $F_s(x)$  is  $H_1(x, H'_1(x)^s)$ .  $l_1$  is a polynomial function of  $\kappa$ .
- HMQV:
  - 1)  $m_2$  is a primer number,  $G_2$  is a  $m_2$ -order cyclic group,  $g_2$  is a generator of  $G_2$ . The length of  $m_2$  is a polynomial function of  $\kappa$ .
  - 2)  $H_2, H'_2$  are two hash functions with ranges  $\mathbb{Z}_{m_2}$  and  $\{0, 1\}^{l_2}$ .  $l_2$  is the length of the session key, which is a polynomial function of  $\kappa$ .
- A honey encryption scheme (Enc, Dec).

### Initialization

- $C$  picks  $s \leftarrow \mathbb{Z}_{m_1}$  as the secret key of  $S$  in 2HashDH<sup>a</sup>, and computes  $rw \leftarrow H_1(pw, H'_1(pw)^s)$ ; computes  $k_U \leftarrow \mathbb{Z}_{m_2}, K_U \leftarrow g_2^{k_U}$  to generate the private/public keys  $(k_U, K_U)$  for  $U$  in HMQV; computes  $c \leftarrow \text{Enc}_{rw}(k_U)$  to yield the ciphertext  $c$  of  $k_U$  using the key  $rw$ ; sends  $(U, s, K_U, c)$  to  $S$ .
- Getting  $(U, s, K_U, c)$  from  $C$ ,  $S$  computes  $k_S \leftarrow \mathbb{Z}_{m_2}, K_S \leftarrow g_2^{k_S}$  to generate the private/public keys  $(k_S, K_S)$  for  $S$  in HMQV; stores  $(U, K_U, s, c)$ .

### Authentication

- $C$  picks  $r \leftarrow \mathbb{Z}_{m_1}$  and computes  $\alpha \leftarrow H'_1(pw)^r$ ; picks  $x \leftarrow \mathbb{Z}_{m_2}$  and computes  $X \leftarrow g_2^x$ ; sends  $(U, X, \alpha)$  to  $S$ .
- Getting  $(U, X, \alpha)$  from  $C$ ,  $S$  picks  $y \leftarrow \mathbb{Z}_{m_2}$  and computes  $Y \leftarrow g_2^y, \beta \leftarrow \alpha^s$ ; sends  $(Y, \beta, c, K_S)$  to  $C$ ; computes  $SK \leftarrow H_2((XK_U^{H'_2(X, K_S)})^{y+H'_2(Y, K_U)k_S})$  and outputs it.
- Getting  $(Y, \beta, c, K_S)$  from  $S$ ,  $C$  computes  $rw \leftarrow H_1(pw, \beta^{1/r})$ ,  $k_U \leftarrow \text{Dec}_{rw}(c)$ ; computes  $SK \leftarrow H_2((YK_S^{H'_2(Y, K_U)})^{x+H'_2(X, K_S)k_U})$  and outputs it.

<sup>a</sup>Note  $s$  can be generated by  $S$ , but this will increase the rounds of the communication between  $C$  and  $S$  to generate  $rw$ .

Fig. 2: A m-OPAQUE

But in practice, as the widely deployed and adopted PKI, many applications use TLS with PKI to establish server-authenticated session keys. This provides stronger authentication, for the server, than PAKE. Thus, PAKE is naturally combined with PKI and TLS to achieve the strong authentication against server impersonation and the password security mentioned above. For example, another standardized PAKE, the Secure Remote Password protocol [30], can be used for TLS Authentication, and this method is also standardized [31] and widely implemented. OPAQUE [12] also provides the flexibility in combination with TLS: 1) running 1-RTT TLS to establish a server-authenticated session key (with  $S$ 's public

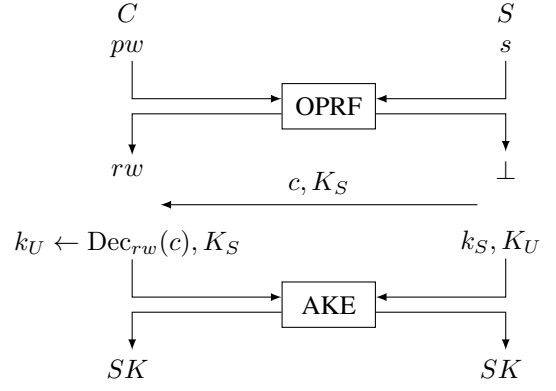


Fig. 3: Schematic diagram of our modified OPAQUE

key distributed by PKI); 2) running OPAQUE to get  $k_U$ ; 3) executing TLS client authentication with  $k_U$ .

To achieve as strong security as possible in reality, we adopt OPAQUE and combine it with PKI to construct our HPAKE. We do not use the above TLS-OPAQUE combination method, because the method adds two rounds of communication (1-RTT TLS) as compared to the original OPAQUE. In contrast, we use the server's public key (distributed by PKI) to sign the server's message (i.e., the message in the second round) in OPAQUE. We do not increase the number of communication rounds but achieve the same security as TLS-OPAQUE, i.e., the strong authentication on the server and the password security. Our method can be seen as running OPAQUE via a server-authenticated channel which is established by the server's public key with the help of PKI. We note that this observation can be easily understood by reader.

We further slightly modify some details of the instantiated OPAQUE to fit our HPAKE construction without compromising the security of OPAQUE. The modified version of OPAQUE (m-OPAQUE) is described in Figure 2. The differences between the original and ours are listed as follows:

- 1) In the original version,  $c$  is the ciphertext of  $k_U, K_S$  encrypted by an authenticated encryption scheme; while in our modification,  $c$  is the ciphertext of  $k_U$  (without  $K_S$ ) encrypted by a honey encryption scheme (without the property of authentication), and  $K_S$  is sent to  $U$  as the plaintext along with  $c$ .
- 2) The original (PKI-free) version uses an unauthenticated channel between  $U$  and  $S$  (in the TLS-OPAQUE version [12],  $U$  and  $S$  first run TLS with PKI to establish a server-authenticated secure channel, then run OPAQUE, and finally perform TLS client authentication); but in our modification,  $U$  and  $S$  run OPAQUE via a server-authenticated channel established with the help of PKI.

We here demonstrate that the modification does not compromise the security.

- 1) About  $K_S$ :

- a) Confidentiality: In our version,  $K_S$  is sent on an authenticated channel, so it can be obtained by the adversary; in the original version, only its ciphertext is sent on the public channel. But this does

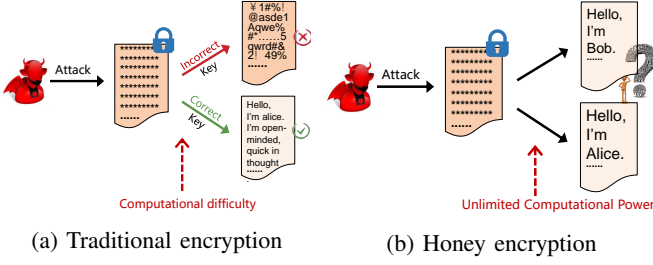


Fig. 4: The difference between traditional encryption and honey encryption

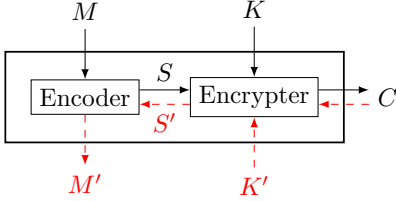


Fig. 5: Construction of honey encryption

not compromise the security, as  $K_S$  doesn't need to be kept secret. The goal of OPAQUE encrypting  $K_S$  is to authenticate  $K_S$  (with an authenticated encryption), rather than making it secret.  $K_S$  is used as  $S$ 's public key in AKE, which usually is a public value. Sending  $K_S$  on the public channel does not compromise the security of AKE and OPAQUE as well.

- b) Authentication: In the original version,  $K_S$  is authenticated by the password; in our version,  $K_S$  is authenticated by the server-authenticated channel. Thus our version can provide a stronger authentication for  $K_S$ .

## 2) About $k_U$ :

- a) Confidentiality: In the original version,  $k_U$  is encrypted by an authenticated encryption scheme; but we assume that  $k_U$  is encrypted by a honey encryption scheme. The modification still keeps the confidentiality of  $k_U$ , because the honey encryption can guarantee the property of confidentiality.
- b) Authentication: Same as  $K_S$ , in the original version,  $k_U$  is authenticated by the password; while we allow  $k_U$  to be authenticated by the server-authenticated channel. Our version thus can provide a stronger authentication for  $k_U$ .

By guaranteeing the confidentiality and authentication of  $k_U$  as well as the authentication of  $K_S$ , our modification keeps the security of AKE and further OPAQUE. The formal proof of our m-OPAQUE can be obtained from the proof of the original OPAQUE with corresponding slight modifications. Since the original proof is complex, we do not provide a detailed proof here, but refer the interested reader to [12].

## C. Honey Encryption

Honey encryption [25], [32] is a novel encryption method, which can yield decoy messages for incorrect keys as shown in Figure 4b. It introduces a probabilistic encoder to encode the message  $M$  to a (fixed-length) uniform bit string  $S$  and then encrypts  $S$  by a carefully-chosen traditional encryption scheme (see Figure 5). The encoder is designed according to the message distribution  $\mathcal{M}$ , which can be uniform or nonuniform (e.g., for the password vaults [33]). The encoder should guarantee that decoding a random bit string will yield a message sampled from  $\mathcal{M}$ . Formally, for an arbitrary adversary (maybe with unlimited computing resources)  $\mathcal{A}$ ,  $(M_0, S_0)$  and  $(M_1, S_1)$  are indistinguishable (we denote  $(M_0, S_0) \sim (M_1, S_1)$ ), where  $S_0 \leftarrow_{\$} \{0, 1\}^l$  (i.e., randomly selecting a  $l$ -bit string),  $M_0 \leftarrow \text{Decode}(S_0)$ ,  $M_1 \leftarrow_p \mathcal{M}$  (i.e., sampling a message from  $\mathcal{M}$  according to the message distribution  $p$ ),  $S_1 \leftarrow \text{Encode}(M_1)$ , and  $l$  is the length of the bit strings. More specifically,

$$|\Pr[\mathcal{A}(M_0, S_0) = 1 : S_0 \leftarrow_{\$} \{0, 1\}^l, M_0 \leftarrow \text{Decode}(S_0)] - \Pr[\mathcal{A}(M_1, S_1) = 1 : M_1 \leftarrow_{\$} \mathcal{M}, S_1 \leftarrow \text{Encode}(M_1)]|$$

is negligible. Please note that the traditional encryption scheme used in honey encryption should yield a random bit string for each incorrect keys. Therefore, for each incorrect key  $K'$ , the honey encryption scheme will produce a  $l$ -bit string  $S'$  and further a plausible-looking message  $M'$  on  $\mathcal{M}$ .

In the design of HPAKE, we use honey encryption to encrypt the user's private key  $k_U$ , which is a uniformly random number on  $\mathbb{Z}_{m_2}$ . Designing an encoder for  $k_U$  is simple. To encode  $k_U$ , we directly select an integer number from  $[\text{round}(k_U 2^l / m_2), \text{round}((k_U + 1) 2^l / m_2)) \subseteq [0, 2^l)$  as  $S$ , where  $\text{round}$  is the rounding function; to decode  $S$ , we find the corresponding interval and obtain  $k_U$ . With the encoder, for each incorrect key  $rw$ , the honey encryption scheme can produce a plausible-looking private key on  $\mathbb{Z}_{m_2}$ .

## D. Becerra et al.'s honeyPAKE

Becerra et al. [34] made an effort to combine PAKE and honeyword techniques to achieve the benefits of both. They called the new protocol model, honeyPAKE. But the model does not capture the security of honeyPAKE in a formal way and it fails to capture the security in multiple sessions, as it only considers the security in a (single) session. Further, as the (main) protocol design is based on a traditional symmetric PAKE - PPK [10], it cannot prevent the insider attacks and pre-computing attacks. More importantly, the design requires the user to remember an additional password (two in all), which doubles the user's memory burden and fails to achieve the expected security because of the user's password reuse habits.

In contrast, we use a game-based model to capture the HPAKE security. The model has been widely adopted in the cryptography and security community to provide sound security analysis approach for protocols. Our model captures the security of HPAKE in multiple sessions, as the classic BR93 model for AKE [35] and BPR2000 model for PAKE [36] do. As for the design, we construct the protocol based on a state-of-the-art aPAKE - OPAQUE. Our design is compatible to the

same user interface as traditional password-only authentication (also as PAKE and honeyword), and does not require the secondary password.

### III. OUR DESIGN

On the top of OPAQUE, honeyword mechanism and honey encryption, we propose an HPAKE construction. The basic idea of our design is to generate honeywords (via a honeyword generation algorithm) and further generate the user  $U$ 's "honey" private/public keys (via honey encryption). If the client  $C$  runs HPAKE with a honeyword, it will get a honey private key and further a honey session key. If the usage of a honey session key occurs, the honeyword mechanism will raise an alarm of password leakage.

#### A. Registration and authentication phases

We give a detailed and formal description of our design in Figure 7. Meanwhile, for ease of understanding, we show the schematic diagram of the design in Figure 6 and briefly describe the process as follows:

- 1) **Registration** (via a secure channel):
  - a)  $C$  runs HPAKE (registration phase) as the same as our modified OPAQUE.
  - b)  $S$  runs HPAKE (registration phase) as the same as our modified OPAQUE and gets the ciphertext  $c$ , OPRF secret  $s$ ,  $S$ 's private key  $k_S$ ,  $U$ 's (real) public key  $K_U$ . Then,  $S$  generates  $t-1$  honeywords, runs OPRF to transform honeywords to honey random passwords, uses the honey random passwords to decrypt  $c$  to get  $U$ 's honey private key, calculates  $U$ 's honey public key, randomly shuffle  $U$ 's real and honey public keys, stores the public key list with  $U, s, c, k_S$  and sends the index/position  $i_r$  of the real public key (with  $U$ ) to  $HC$ .
  - c)  $HC$  stores  $U, i_r$ .
- 2) **Authentication** (via a  $S$ -authenticated secure channel. Note the channel is established by 1-RTT TLS protocol.):
  - a)  $C$  runs HPAKE (authentication phase) as the same as our modified OPAQUE to get the session key  $SK$ . Then,  $C$  uses  $SK$  for further data transmission.
  - b)  $S$  runs HPAKE (authentication phase) as the same as our modified OPAQUE for  $U$ 's private key to get  $t$  session keys  $\{SK_i\}_{i=1}^t$ . If  $C$  uses  $SK$ ,  $S$  checks if  $SK \in \{SK_i\}_{i=1}^t$ :
    - i) If it is the  $i$ -th one, send  $(U, i)$  to  $HC$ .
    - ii) Otherwise, deny this access.
  - c)  $HC$  checks if  $i = i_r$ :
    - i) If it is, allow this access.
    - ii) Otherwise, raise an alarm.

#### B. Security Analysis

Here, we briefly discuss the security of our HPAKE, and in Section IV, we will give a formal analysis.

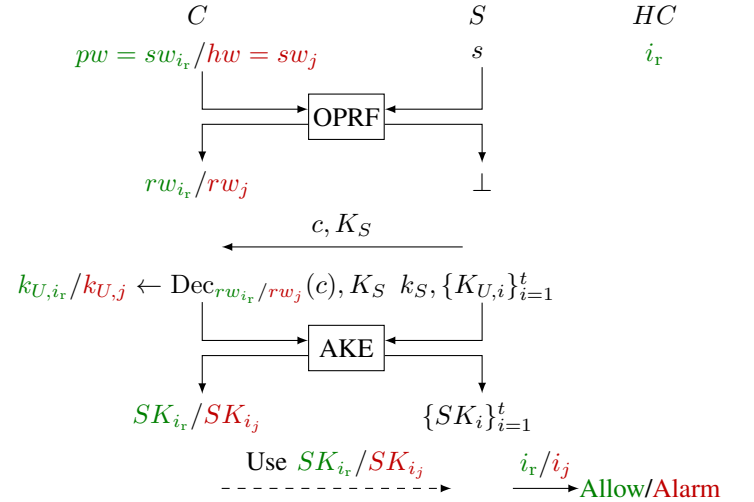


Fig. 6: Schematic diagram of our HPAKE. Here,  $x/y$  represents  $x$  or  $y$ , not the ratio. Besides, the processes including OPRF, AKE, and the sending of  $c, K_S$  are running parallel, not sequentially.

**Resisting online password guessing.** In our HPAKE, the adversary can impersonate the client to carry out online password guessing, which is the same as in aPAKE. In this case (where the adversary does not compromise the authentication server's storage file), the interface of our HPAKE is the same as that of OPAQUE for the adversary. Therefore, our design achieves the same level of security as OPAQUE and password-over-TLS against online password guessing.

**Resisting offline password guessing and detecting password leakage.** We allow the authentication server to store a user  $U$ 's secret key  $s$  for OPRF, the ciphertext  $c$  of  $U$ 's private key and the sweet public keys  $\{K_{U,i}\}_{i=1}^t$ . If the adversary compromises the authentication server, she can carry out offline password guessing to recover the password. To this end, she needs to 1) run OPRF to get the corresponding random password  $rw'$  for each password guess  $pw'$  (in the dictionary), 2) decrypt the ciphertext  $c$  with  $rw'$  to get the corresponding private/key  $k'_U, K'_U$ , and 3) check if  $K'_U$  is in the sweet public keys  $\{K_{U,i}\}_{i=1}^t$ . If it is, then the guess is a sweetword. Accordingly, the adversary gets  $t$  sweetwords by offline password guessing. If the honeyword generation algorithm is ideal, then the adversary cannot tell which sweetword is the real one. To compromise  $U$ 's account, she probably runs HPAKE with a honeyword. In this case, the session key established by the adversary is not a real one, and the honeychecker will identify it and raise an alarm. From the above descriptions, we can see that the HPAKE can resist offline password guessing and detect the password leakage even if the authentication server is compromised, which is as secure as the password-over-TLS with honeyword mechanism, but more secure than aPAKE.

**Resisting insider attacks and malicious server.** The password used by the client will not be directly sent to the authentication server, so that the insider or malicious server cannot steal the password plaintext. But the online password

guessing against aPAKE can be still carried out. In this case, our HPAKE achieves the same level of security as aPAKE, but is more secure than those schemes safeguarded by password-over-TLS.

### C. Discussions

**The need of a server-authenticated channel.** The server-authenticated channel is of extreme necessity for HPAKE. Otherwise, HPAKE cannot reliably detect the password leakage. We demonstrate this point with the following attack. Targeting an HPAKE without the server-authenticated channel, the adversary can 1) compromise the authentication server, 2) get the storage file, 3) impersonate the authentication server with the file to run HPAKE with the real user. She can obtain the index  $i_r$ , and further get the real password by carrying out offline password guessing. To resist the attack, we should require a server-authenticated channel.

**Usage of honey encryption.** With honey encryption, we do not need to run several aPAKE instances, but only one. The client can run the instance with a honeyword, so that the honey encryption will yield a  $U$ 's honey private key  $k'_U$ . Using the private key, the client will obtain a honey session key  $SK'$ . Further, one uses this session key that will lead to an alarm of password leakage.

We do not use honey encryption to encrypt  $S$ 's public key  $K_S$ , in order to avoid a situation that we cannot get the corresponding  $S$ 's private key  $k_S$  to run AKE. More specifically, in the initialization phase, if  $K_S$  is encrypted by honey encryption, then decrypting the ciphertext (with a honey random password  $rw'$ ) will yield a honey public key  $K'_S$ . But the decryption does not produce the corresponding private key  $k'_S$ , which is needed for the server to run AKE (i.e., calculating the session key).

**Number of honeywords.** Intuitively, if we increase the number of honeywords in the system, the attacker may have less probability to identify the real password. But a large number of honeywords may increase the storage and computation cost significantly. One may balance the tradeoff among usability, practicability and security. Juels and Rivest [15] thus recommended to practically set the number to 20, for general purpose. It could be the case that different users, based on privilege or weight, may be assigned with various values, e.g., 200 for VIP, and 2 for basic subscribers. But on average setting 20 sweetwords (on average) per account is sufficient for a system (which may include thousands or even millions accounts). We note that if the attacker tries to log in some accounts with honeywords, the honeyword mechanism will raise alarms for the whole password file (targeting to all the accounts, instead of those under attacks). In this way, all the users can be notified about the threat.

Suppose there exists a website that includes 10,000 accounts. In traditional password-over-TLS (without any honeyword mechanisms), the attacker can compromise almost all accounts (if there is no other security protection), considering the effectiveness of current password guessing algorithms [3]. Applying a honeyword mechanism, one may set 20 as the number of sweetwords and 1,000 as the threshold of honey-

word logins (i.e., if the number of honeyword logins exceeds 1,000, the system will take correspond actions, e.g., shutting down the system). Based on these numbers, the attacker can only compromise 50 (1000/20) accounts, on average, which is a very small fraction (0.5%) of the 10,000 accounts.

## IV. SECURITY ANALYSIS FOR OUR DESIGN

To formally analyze the security of HPAKE, we propose a game-based security model and perform the security analysis of our design in the model.

### A. Security Model

Our security model is a game-based one inspired by BPR2000 model for PAKE [36] and the CK-adversary model for AKE [37]. Game-based security models get some criticism (e.g., suffering with overlooking corner cases [38] and at times failing to get secure composition [39]), and many automatic analysis methods/tools (e.g., Smartverif [40], CryptHOL [38]) are proposed to give comprehensive analysis and avoid human errors in proofs. But so far, the automatic methods cannot provide a cryptography sound security analysis, game-based models are still the mainstream security analysis method, especially for AKE [35] and its variants (e.g., PAKE [36], multi-factor AKE [20]). Hence, we choose this type of security model for our HPAKE.

**Protocol participants and communication model.** In PAKE, there are two parties: a client  $C$  and a (authentication) server  $S$ . And  $C$  is held by a user  $U$ . For authentication,  $U$  needs to enter the username  $U$  and the password  $pw$  on  $C$ . Like PAKE, HPAKE shares the same parties but an extra one is required: an enhanced server - honeychecker  $HC$ .

The communication between  $S$  and  $HC$  is on a secure channel, which may be on the external network with a pre-established session key or on the internal network. In the registration phase, the communication between  $C$  and  $S$  is on a secure channel. In practice, this registration usually is done via a secure TLS channel or even a face-to-face approach. In the authentication, the communication between  $C$  and  $S$  is via a server-authenticated channel as discussed before.

**Protocol execution.** In the registration phase of an HPAKE  $\Pi$ , the user  $U$  registers its username ( $U$ ) with a password  $pw$  on the client  $C$  and meanwhile, the authentication server  $S$  generates honeywords and stores the password file (which includes the real password  $pw$  and  $t - 1$  honeywords), and the honeychecker  $HC$  stores the index/position  $i_r$  of the real password  $pw$  in the password list. Note there is no need to register the client  $C$  held by the user. Any user can leverage any client for the registration or authentication. A user can even leverage different clients for HPAKE instances. And a client can be used by various users in different time slots. This brings the same level of usability as the password-based authentication.

In the authentication phase of  $\Pi$ , the user  $U$  inputs the username  $U$  and the password  $pw$  on the client  $C$ , in which the client may be different from the one used in the registration phase. After the interactions between  $C$  and  $S$  via the server-authenticated channel,  $C$  yields a session key  $SK$  and  $S$

### Parameter

- Security parameter  $\kappa$ .
- 2HashDH:
  - 1)  $m_1$  is a primer number,  $G_1$  is a  $m_1$ -order cyclic group, and  $g_1$  is a generator of  $G_1$ . The length of  $m_1$  is a polynomial function of  $\kappa$ .
  - 2)  $H_1, H'_1$  are two hash functions with ranges  $\{0, 1\}^{l_1}$  and  $G_1$ , respectively. The PRF  $F_s(x)$  is  $H_1(x, H'_1(x)^s)$ .  $l_1$  is a polynomial function of  $\kappa$ .
- HMQV:
  - 1)  $m_2$  is a primer number,  $G_2$  is a  $m_2$ -order cyclic group,  $g_2$  is a generator of  $G_2$ . The length of  $m_2$  is a polynomial function of  $\kappa$ .
  - 2)  $H_2, H'_2$  are two hash functions with ranges  $\mathbb{Z}_{m_2}$  and  $\{0, 1\}^{l_2}$ .  $l_2$  is the length of the session key, which is a polynomial function of  $\kappa$ .
- A honey encryption scheme (Enc, Dec) for the message space  $\mathbb{Z}_{m_2}$ .
- A honeyword generation algorithm Gen.

### Initialization (via a secure channel)

- The client  $C$  picks  $s \leftarrow \mathbb{Z}_{m_1}$  as the secret key of  $S$  in 2HashDH, and computes  $rw \leftarrow H_1(pw, H'_1(pw)^s)$ ; computes  $k_U \leftarrow \mathbb{Z}_{m_2}$ ,  $K_U \leftarrow g_2^{k_U}$  to generate the private/public keys  $(k_U, K_U)$  for  $U$  in HMQV; computes  $c \leftarrow \text{Enc}_{rw}(k_U)$  to yield the ciphertext  $c$  of  $k_U$  using the key  $rw$ ; sends  $(U, s, K_U, c)$  to  $S$ .
- Getting  $(U, s, K_U, c)$  from the client  $C$ , the authentication server  $S$  computes  $k_S \leftarrow \mathbb{Z}_{m_2}$ ,  $K_S \leftarrow g_2^{k_S}$  to generate its private/public keys  $(k_S, K_S)$  in HMQV;  $S$  generates  $t - 1$  honeywords  $hw_i \leftarrow \text{Gen}$  for  $i$  from 1 to  $t - 1$ , the corresponding random honeyword  $rw_i \leftarrow H_1(pw, H'_1(hw_i)^s)$ , and the honey private/public keys  $k_{U,i} \leftarrow \text{Dec}_{rw_i}(c)$ ,  $K_{U,i} \leftarrow g_2^{k_{U,i}}$ ; randomly shuffles  $K_{U,i}$  ( $1 \leq i \leq t - 1$ ) with  $K_U$ , and sends the index  $i_r$  of the real one (with the ID  $U$ ) to the honeychecker  $HC$ ; stores  $s, c$  with the shuffled  $K_{U,i}$  ( $1 \leq i \leq t$ ).
- Getting  $(U, i_r)$  from the authentication server  $S$ , the honeychecker  $HC$  stores it.

### Authentication (via a server-authenticated channel)

- $C$  picks  $r \leftarrow \mathbb{Z}_{m_1}$  and computes  $\alpha \leftarrow H'_1(pw)^r$ ; picks  $x \leftarrow \mathbb{Z}_{m_2}$  and computes  $X \leftarrow g_2^x$ ; sends  $(U, X, \alpha)$  to  $S$ .
- Getting  $(U, X, \alpha)$  from  $C$ ,  $S$  picks  $y \leftarrow \mathbb{Z}_{m_2}$  and computes  $Y \leftarrow g_2^y$ ,  $\beta \leftarrow \alpha^s$ ; sends  $(Y, \beta, c, K_S)$  to  $C$ ; computes  $SK_i \leftarrow H_2((XK_{U,i}^{H'_2(X, K_S)})^{y+H'_2(Y, K_{U,i})k_S})$  for  $i$  from 1 to  $t$ .
- Getting  $(Y, \beta, c, K_S)$  from  $S$ ,  $C$  computes  $rw \leftarrow H_1(pw, \beta^{1/r})$ ,  $k_U \leftarrow \text{Dec}_{rw}(c)$ ; computes  $SK \leftarrow H_2((YK_S^{H'_2(Y, K_U)})^{x+H'_2(X, K_S)k_U})$  and further uses  $SK$  for data transmission (or other purposes).
- $S$  checks if the session key  $SK$  used by  $C$  is one of  $\{SK_i\}_{i=1}^t$ :
  - 1) If it is not, deny this session.
  - 2) If it is the  $i$ -th one,  $S$  sends  $(U, i)$  to  $HC$  (via a secure channel). Then  $HC$  checks if the index  $i$  is correct for  $U$  (i.e., equal to  $i_r$ ):
    - a) If it is, allow this session.
    - b) Otherwise, raise an alarm of password leakage and take actions according to the pre-defined security policy.

Fig. 7: Our HPAKE Construction

outputs several session keys  $\{SK_i\}_{i=1}^t$  including a real and several honey keys. Then,  $C$  uses its session key  $SK$  for further data transmission, And  $S$  can tell if  $SK$  is in  $\{SK_i\}_{i=1}^t$ . If not, then  $S$  denies the session. If it is the  $i$ -th one,  $S$  sends  $i$  to the honeychecker  $HC$ . Further if  $i = i_r$ ,  $HC$  allows the session and otherwise, raises an alarm of password leakage.

To capture the security of  $\Pi$  in the context of parallel running, our model allows the parties to parallel run more than one instance of  $\Pi$ . In order to distinguish, we denote the  $i$ -th instance of a party  $P$  as  $P^i$ .

**Partnering instances.** To capture the partnering of the parties' instances, our model uses session id (SID). We require  $C$  and  $S$  to establish a SID before running HPAKE, and the communication between  $S$  and  $HC$  inherits the SID. In practice,  $C$  and  $S$  can send two random number  $n_1, n_2$  to

each other, separately, and uses  $C||S||n_1||n_2$  as SID. If the party instances  $C^i$  and  $S^j$  use the same SID, we say  $C^i$  and  $S^j$  are *partners*.

**Attacker ability.** The attacker can eavesdrop the messages between  $S$  and  $C$ , delay and replay the messages from  $S$  to  $C$ , but can intercept, tamper and forgery the messages from  $C$  to  $M$ . Note the communication between  $S$  and  $C$  is on a  $S$ -authenticated channel.

Our model allows the attacker to corrupt  $C$ , i.e., getting all internal states of  $C$  and further fully controlling  $C$ . This models the case where the attacker fully controls  $C$  via virus or Trojan Horse. Our model also allows the attacker to steal the user's password  $pw$  without corrupting  $C$ , capturing the case that the attacker may obtain  $pw$ , e.g., via over-shoulder attacks [41].

For the authentication server  $S$  and the honeychecker  $HC$ ,

our model only allows the attacker to steal the password file on  $S$ , but does not allow she to corrupt  $S$  or  $HC$ , or steal the storage file on  $HC$ . The former case where the attacker steals the password file on  $S$  is the main threat and the motivation for HPAKE. The latter cases where  $S$  or  $HC$  is corrupted, or the storage file on  $HC$  is stolen is not considered, because the security in these cases is trivial and considering them will lead to the unexpected and unnecessary complexity of the security model. Here, we briefly discuss the unconsidered cases:

- 1) If  $S$  is corrupted, the attacker can tell the real password (see the discussion on the server-authenticated channel in Section III-C) and compromise the account.
- 2) If the storage of  $HC$  is compromised or  $HC$  is corrupted, the attacker cannot get any information about the password except she steals the password file.
- 3) In other combined cases (e.g., both  $S$  and  $HC$  is corrupted), the attacker can do what she can in each single case.

Our model allows the attacker to reveal a (used) session key (note that for  $S$ , this is the real session key rather than a honey one). In practice, the session key may not be safely and fully destroyed after its usage, and may be obtained by the attacker. A secure HPAKE should guarantee the security of the rest of the session keys even if some of them are revealed.

**Security game.** Same as BPR2000 model for PAKE [36], our model is game-based. Usually in a game-based security model, there is a challenger and an attacker. The challenger simulates the running of the protocol and the attacker takes the actions by sending queries to the challenger. The queries formally capture the attacker's ability. The attacker wins the game, if she breaks the protocol. The security of the protocol is defined by the advantage (probability) of the attacker winning the game. More specifically, the protocol is defined to be secure, if no attacker can win the game with an advantage more than a given bound. The given bound is defined by the advantage of the trivial/unavoidable attacker.

For an HPAKE  $\Pi$ , compromising the session keys is the attacker's goal. To test if the attacker knows the session key or the partial information about the session key, we require the attacker to distinguish the session key from a random number (with the same length). If she can tell, then she wins the game, i.e., breaking  $\Pi$ .

According to the attacker's ability as discussed before, our model allows the attacker to make the following queries and lets the challenger respond as follows.

- 1) **Send**( $S, i, C, M$ ): Execute  $\Pi$  as the instance  $S^i$  of  $S$  getting the message  $M$  from  $C$ , and respond the response message of  $S^i$  to the attacker.  
Note if  $M$  is not a message responded by the challenger as the last message from  $S^i$  to  $C$ , we say that this **Send** query is *rogue*. In other words, a **Send** query is *rogue*, meaning the message is tampered or forged by the attacker.
- 2) **Send**( $C, i, S, M$ ): If  $M$  is responded as a message from  $S$  before, execute  $\Pi$  as the instance  $C^i$  of  $C$  getting the message  $M$  from  $S$ , and respond the response message of  $C^i$  to the attacker.

- 3) **Send**( $C, i, S, \perp$ ): If  $C^i$  does not exist, initialize the instance  $C^i$  of  $C$ , run  $\Pi$  as  $C^i$ , and respond the initial message from  $C^i$  to  $S$ .

In the real world, the attacker usually cannot make the client initiate a session. But in the game, this query allows the attacker to initiate sessions as much as she wants. This setting is used to simulate the parallel running of  $\Pi$ .

- 4) **Use**( $C, i, S, SK$ ): If  $SK$  is not one of the session keys for the session between  $C^i$  and  $S$ , respond Deny; else if  $SK$  is the real session key, respond Allow; otherwise, stop the game, meanwhile the attacker fails the game.  
Note if  $SK$  is not obtained by the **Reveal**( $C, i$ ) or **Reveal**( $S, j$ ) query ( $S^j$  is the partner of  $C^i$ ), we say this **Use**( $C, i, S, SK$ ) query is *rogue*.
- 5) **Reveal**( $P, i$ ): Respond the session key of  $P^i$  if it has. Note if  $P = S$ , the session key is real.
- 6) **Corrupt**( $C$ ): Respond the internal states and the passwords of all instances of  $C$  (so that the attacker can impersonate  $C$ ).
- 7) **StealPW**( $U$ ): Respond the password  $pw$  of  $U$ .
- 8) **StealPWFile**( $S, U$ ): Respond the  $U$ 's password file on  $S$ .
- 9) **Test**( $P, i$ ): If no **Test**( $\cdot, \cdot$ ) was made before,  $P^i$  has established a session key and the key is *fresh*, then flip a coin  $b$ .
  - a) If  $b = 1$ , respond the (real) session key of  $P^i$ ;
  - b) If  $b = 0$ :
    - i) If  $P = S$ , **StealPWFile**( $S, U$ ) was made, and at least one *rogue* **Send**( $S, i, C, \cdot$ ) query was made, then randomly select one from the honey session keys of  $S^i$  and respond it.
    - ii) Otherwise, randomly select a key in the session key space (note the space usually is  $\{0, 1\}^l$  where  $l$  is the length of the session key) and respond the key.

**Advantage.** The **Test** query is used to capture the attacker's advantage (rather than ability). Given the real session key or the random key (or a honey session key), the attacker needs to make a guess  $b'$  on  $b$ . If  $b' = b$ , the attacker wins the game. Traditionally, we define the advantage of  $\mathcal{A}$  as

$$\text{Adv}_{\Pi}^{\text{hpake}}(\mathcal{A}) = |\Pr[b' = 1|b = 0] - \Pr[b' = 1|b = 1]|.$$

This definition captures the advantage of  $\mathcal{A}$  getting the partial information of the real session key  $SK$ . If an attacker has an advantage of 1, then she knows the full information of  $SK$  (and can use it for further impersonation), since she can always tell  $SK$  from a random or honey session key. If an attacker has a negligible advantage to distinguish  $SK$  from a random session key, then she knows nothing about  $SK$  and has a negligible probability of impersonation or eavesdropping in the session with  $SK$ . If an attacker has a negligible advantage to distinguish  $SK$  from a honey session key (when  $P = S$ , **StealPWFile**( $S, U$ ) was made, and at least one *rogue* **Send**( $S, i, C, \cdot$ ) query was made), then she knows nothing about the index  $i_r$  of the real password (or

session key), but she may have  $\frac{1}{t}$  probability of impersonation or eavesdropping in the session since she can retrieve  $t$  sweetwords from the password file.

**Freshness of the session keys.** The attacker knows the session key via trivial methods, e.g., **Reveal** or **Corrupt** queries. These session keys cannot be guaranteed secure by any HPAKE. Excluding these (unfresh) session keys, our model only allows fresh session keys to be made in **Test** queries. Formally, we have the following definition.

**Definition 1** (Freshness). The session key of an instance  $P^i$  is *fresh*, if **Reveal**( $P, i$ ) and **Reveal**( $Q, j$ ) were not made, where  $Q^j$  is the partner instance of  $P^i$  (if it exists), and one of the following conditions holds:

- 1) None of the **Corrupt**( $C$ ) or **StealPW**( $U$ ) queries was made.
- 2) The internal states of  $P^i$  and  $Q^j$  are not sent to the attacker, and no rogue **Send**( $S, i, C, \cdot$ ) queries were made.

Note that if the **Corrupt**( $C$ ) or **StealPW**( $U$ ) queries were issued, the attacker knows the password, so that she can impersonate  $U$  to establish the session keys with  $S$ . In this case, the keys in these sessions are unsafe and the internal states of the sessions are known to the attacker. But the keys in the following sessions are still safe:

- 1) The sessions established before the **Corrupt**( $C$ ) or **StealPW**( $U$ ) queries were made.
- 2) The sessions established by the real user and not tampered by the attacker.

The former corresponds to the forward security, while the latter is for the key-compromise impersonation (KCI) security/resistance.

**Security definition.** With the above modelling, we can finally give the formal definition of the security of HPAKE.

**Definition 2** (HPAKE). An HPAKE  $\Pi$  is secure, if for a uniform password distribution on a dictionary of size  $n$ , an arbitrary probabilistic polynomial time (PPT) attacker  $\mathcal{A}$ , and the security parameter  $\kappa$ , the advantage of  $\mathcal{A}$  is bounded as follows:

- 1) If **StealPWFile**( $S, U$ ) was made,

$$\text{Adv}_{\Pi}^{\text{hpake}}(\mathcal{A}) \leq \text{negl}(\kappa),$$

where  $t$  is the number of the sweetwords for an account, and  $\text{negl}(\kappa)$  denotes a negligible amount in  $\kappa$ .

- 2) Otherwise,

$$\text{Adv}_{\Pi}^{\text{hpake}}(\mathcal{A}) \leq \frac{1}{n}q_S + \text{negl}(\kappa),$$

where  $q_S$  is the number of rogue **Send** queries made by  $\mathcal{A}$ .

**Explanation on the bounds.**

- 1) For the first case where the attacker steals the password file (via the **StealPWFile**( $S, U$ ) query), she can retrieve the plaintext of the  $t$  sweetwords via offline guessing. But for a secure HPAKE, she should not tell the real password from the  $t$  sweetwords.

- a) If the attacker uses one sweetword to carry out on-line guessing (via rouge **Send**( $S, i, C, \cdot$ ) queries), she cannot tell the real session key. So for the **Test**( $S, i$ ) query responding a real session key or a honey one, she has negligible advantage to distinguish the key.
- b) If the attacker does not carry out active attacks (i.e., not making rouge **Send**( $S, i, C, \cdot$ ) queries), she knows nothing about the real session key and the honey ones. So for the **Test**( $S, i$ ) query responding a real session key or a random one, she cannot tell which one is real.

- 2) For the second case where **StealPWFile**( $S, U$ ) was not made, HPAKE has the same security as PAKE. The trivial attacker can carry out online guessing to establish a session key with  $S$ . For a uniform password distribution with the space of size  $n$ , each guess only has the probability of  $1/n$  to succeed. This defines the bound for this case. Although the assumption of password uniform distributions is widely used in password cryptography, some statistic studies on passwords [42], [24], [3] show that the password distribution is far from the uniform distribution. Fortunately, for a non-uniform password distribution with the cumulative distribution function  $f$ , we can simply modify the bounds to suit the password distribution, via replacing  $\frac{1}{2n}q_C$  in the bound of the second case with  $\frac{1}{2}f(q_C)$ .

## B. Security Proof

It is easy to prove the security of our HPAKE in the second case (where **StealPWFile**( $S, U$ ) was not made), since it inherits the security of OPAQUE. As for the security analysis in the first case, it may be challenging. For this case, we first give a lemma to simply the proof.

**Lemma 1.** *If an HPAKE  $\Pi$  is secure for  $t = 2$ , then  $\Pi$  is secure for arbitrary  $t \geq 2$ , where  $t$  is the number of the sweetwords for an account.*

*Proof.* For convenience, we denote  $\Pi^t$  as  $\Pi$  with the sweetword number  $t$ ,  $G^t$  as the game with  $\Pi^t$  and  $\mathcal{A}^t$  as the attacker for  $\Pi^t$ .

In the second case (where  $\mathcal{A}$  did not make **StealPWFile**( $S, U$ )),  $G^2$  is the same as  $G^t$  ( $t \in \mathbb{Z}$ ). Thus

$$\text{Adv}_{\Pi^t}^{\text{hpake}}(\mathcal{A}) = \text{Adv}_{\Pi^2}^{\text{hpake}}(\mathcal{A}) \leq \frac{1}{n}q_S + \text{negl}(\kappa).$$

In the first case, if  $\mathcal{A}$  do not make rouge **Send**( $S, i, C, \cdot$ ) queries,  $\mathcal{A}$  cannot know any partial information of the real session key, i.e.,

$$\text{Adv}_{\Pi^t}^{\text{hpake}}(\mathcal{A}) = \text{Adv}_{\Pi^2}^{\text{hpake}}(\mathcal{A}) \leq \text{negl}(\kappa).$$

Besides, if  $\mathcal{A}$  makes **Test**( $U, i$ ) queries, she only has a negligible advantage, since she cannot make rouge **Send**( $C, i, S, \cdot$ ) queries. So we only need to consider the **Test**( $S, i$ ) query.

In the following, we consider the case where  $\mathcal{A}$  steals password file and makes at least one rouge **Send**( $S, i, C, \cdot$ )

query. In this case, the only difference between  $G^2$  and  $G^t$  ( $t \in \mathbb{Z}$ ) is:

- 1) In  $G^2$ ,  $\mathcal{A}^2$  needs to tell a real session from a decoy one.
- 2) In  $G^t$ ,  $\mathcal{A}^t$  needs to tell a real session from  $t - 1$  decoy ones. More specifically, the real session key has  $\frac{1}{2}$  probability to be responded by  $\text{Test}(S, i)$  and each honey one has  $\frac{1}{2(t-1)}$  probability.

Intuitively, if a real session key is indistinguishable from a decoy one, then it is indistinguishable from  $t - 1$  decoys. Formally, we do the following reduction. From the attacker  $\mathcal{A}^t$  and the game  $G^2$ , we construct the following game  $G^{t*}$  and the attacker  $\mathcal{A}^2$ .

- 1) For  $G^2$  with two sweetwords  $sw_1, sw_2$ , we run the initialization of HPAKE to generate other honeywords  $sw_3, sw_4, \dots, sw_t$  and randomly shuffle  $\{sw_i\}_{i=1}^n$ . Denote the subscripts of the original two sweetwords in the new list as  $i_1, i_2$ , respectively.
- 2) When  $\mathcal{A}^t$  makes the  $\text{Test}(S, i)$  query, randomly select one from  $\{SK_{i_1}, SK_{i_2}\}$  and respond it. (Note in  $G^t$ , the honey session key is randomly selected from all honey session key.)
- 3) Run  $\mathcal{A}^t$  to output the guess  $b'$  for  $b$ , then  $\mathcal{A}^2$  outputs  $b'$  as her guess.

Then  $\mathcal{A}^2$  wins in  $G^2$  if and only if  $\mathcal{A}^t$  wins in  $G^{t*}$ . Besides,  $\{sw_i\}_{i=1}^n$  are randomly shuffled in  $G^{t*}$ , which is equivalent to that a honey session key is randomly selected from  $t - 1$  ones in  $G^t$ . Thus,

$$\Pr(\mathcal{A}^2 \text{ wins in } G^2) = \Pr(\mathcal{A}^t \text{ wins in } G^{t*}) = \Pr(\mathcal{A}^t \text{ wins in } G^t).$$

Then

$$\text{Adv}_{\Pi^t}^{\text{hpake}}(\mathcal{A}^t) = \text{Adv}_{\Pi^2}^{\text{hpake}}(\mathcal{A}^2).$$

If  $\text{Adv}_{\Pi^t}^{\text{hpake}}(\mathcal{A}^t)$  is non-negligible in  $\kappa$ , then  $\text{Adv}_{\Pi^2}^{\text{hpake}}(\mathcal{A}^2)$  is non-negligible in  $\kappa$ . This means if  $\Pi^t$  is not secure, then  $\Pi^2$  is insecure. Therefore, if  $\Pi^2$  is secure, then  $\Pi^t$  is also secure.  $\square$

Then we give the security proof for our HPAKE.

**Theorem 1.** *Our HPAKE in Figure 7 is secure, if OPAQUE is secure and the honeyword generation algorithm is secure (i.e., the honeyword is indistinguishable from the real password).*

*Proof.* In the second case (where  $\mathcal{A}$  did not make  $\text{StealPWFile}(S, U)$ ), if  $\mathcal{A}$  breaks our HPAKE, then she breaks OPAQUE. Thus, according to the security of OPAQUE,

$$\text{Adv}_{\Pi}^{\text{hpake}}(\mathcal{A}) \leq \frac{1}{n}q_S + \text{negl}(\kappa).$$

For the first case, by Lemma 1, we only need to prove the security for  $t = 2$ . In this case, if no **Reveal** queries are made, then  $\mathcal{A}^2$  knows nothing to tell the real session key (or password) from the honey one; but if a **Reveal** query is made,  $\mathcal{A}^2$  may leverage this used real session key to tell the target password or session key. In the following, we will show that  $\mathcal{A}^2$  still cannot tell the real session except she solves the computational Diffie-Hellman problem.

The only way for  $\mathcal{A}^2$  to tell the real session key is to tell the real password and run HPAKE with it. Otherwise,

$\mathcal{A}^2$  cannot get any information about the target real session key (treating  $H'_2$  as a random oracle). The only information about the real password that  $\mathcal{A}^2$  can get is the used real session key(s). A real session key is calculated with  $U$ 's real private/public key (denoted as  $k_{U,r}, K_{U,r}$ ), and meanwhile the honey one is calculated with the honey real private/public key (denoted as  $k_{U,h}, K_{U,h}$ ). More, specifically, the real session key is  $H_2((XK_{U,r}^{H'_2(X, K_S)})^{y+H'_2(Y, K_{U,r})k_S})$  in  $S$  or  $H_2((YK_S^{H'_2(Y, K_{U,r})})^{x+H'_2(X, K_S)k_{U,r}})$  in  $U$ , and the honey one is  $H_2((XK_{U,h}^{H'_2(X, K_S)})^{y+H'_2(Y, K_{U,h})k_S})$  or  $H_2((YK_S^{H'_2(Y, K_{U,h})})^{x+H'_2(X, K_S)k_{U,h}})$ . Given a revealed real session key,  $\mathcal{A}^2$  needs to calculate  $(XK_{U,i}^{H'_2(X, K_S)})^{y+H'_2(Y, K_{U,i})k_S} = (YK_S^{H'_2(Y, K_{U,i})})^{x+H'_2(X, K_S)k_{U,i}}$  ( $i = 0, 1$ ) to verify which  $k_{U,i}$  ( $sw_i$ ) is real. This is equivalent to calculating  $X^y = Y^x$  from  $X$  and  $Y$ , i.e., solving the computational Diffie-Hellman (CDH) problem. Therefore, in this case,

$$\text{Adv}_{\Pi^2}^{\text{hpake}}(\mathcal{A}^2) \leq \text{Adv}^{\text{cdh}},$$

where  $\text{Adv}^{\text{cdh}}$  is the max advantage for CDH problem. Under the CDH assumption (which is also needed for the security of OPAQUE and HMQV),  $\text{Adv}^{\text{cdh}} \leq \text{negl}(\kappa)$ , and further

$$\text{Adv}_{\Pi^2}^{\text{hpake}}(\mathcal{A}^2) \leq \text{negl}(\kappa).$$

To summarize, the advantage of attackers is bounded as in Definition 2. This concludes our HPAKE is secure.  $\square$

## V. EFFICIENCY EVALUATION

To evaluate the efficiency of our HPAKE, we implement a prototype for our HPAKE and deploy it in the real-world environment. Since OPAQUE is recommended by the Crypto Forum Research Group (CFRG) of IETF, there appear many open-source implementations of OPAQUE and two of them [43], [44] are selected to be shown on the page of CFRG [26]. Since the implementation in Rust [44] is built with the help of Hugo Krawczyk (one of the authors of the OPAQUE paper [12]), we leverage it to implement our HPAKE.

### A. Implementation for our HPAKE

Compared with HMQV, our HPAKE extra requires the server-authenticated channel, the generation of honeywords and honey session keys, the honeychecker, and the secure channel between the authentication server and honeychecker. For the server-authenticated channel, we use the Elliptic Curve Digital Signature Algorithm (ECDSA) as the signature scheme. For the generation of honeywords, we simply use the sampling-from-a-probability-model method [16]. Note that, our HPAKE does not allow the server to get the password plaintext, so that the real-password-based generation methods, e.g., chaffing-by-tail-tweaking, do not apply. For the generation of honey session keys, we only need to run OPAQUE with the honeywords. For the honeychecker, we use another server. For the secure channel between the authentication server and honeychecker, we pre-set a symmetric key for the authentication server and honeychecker to encrypt their communication.

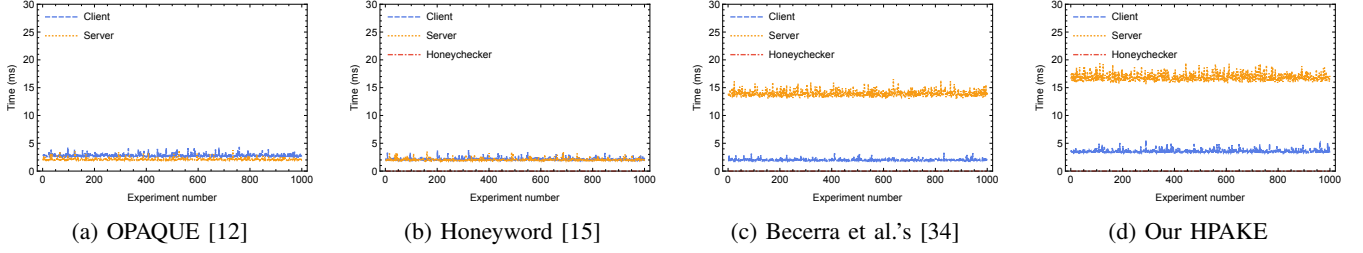


Fig. 8: Computational cost comparison

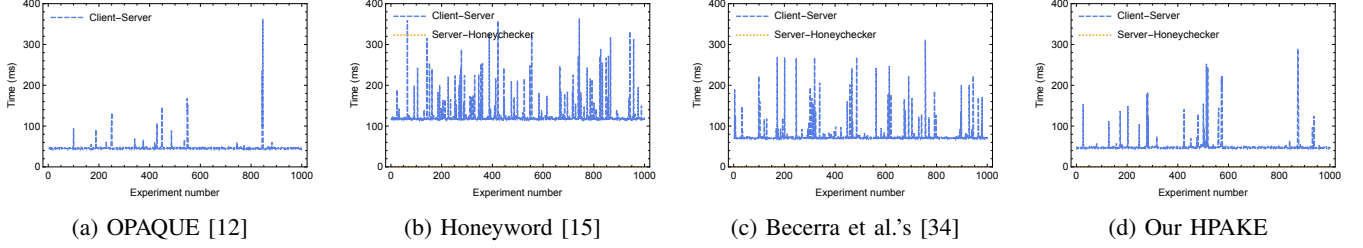


Fig. 9: Communication cost comparison

TABLE II: Performance summary

| Protocol              | Computation cost          |                              |                  | Storage cost |          |              | Communication cost    |                     |
|-----------------------|---------------------------|------------------------------|------------------|--------------|----------|--------------|-----------------------|---------------------|
|                       | Client                    | Server                       | Honeychecker     | Client       | Server   | Honeychecker | Client-Server         | Server-Honeychecker |
| OPAQUE [12]           | 3 exp + 1 mexp<br>2.72 ms | 2 exp + 1 mexp<br>2.07 ms    | -<br>-           | 0 bit        | 704 bit  | -            | 2 rounds<br>47.01 ms  | -<br>-              |
| Honeyword [15]        | 2 exp + 1 mexp<br>2.07 ms | 3 exp<br>2.00 ms             | 0 exp<br>0.00 ms | 0 bit        | 5184 bit | 96 bit       | 5 rounds<br>125.88 ms | 2 rounds<br>0.12 ms |
| Becerra et al.'s [34] | 2 exp<br>1.37 ms          | 1 + $t$ exp<br>14.34 ms      | 0 exp<br>0.00 ms | 0 bit        | 5376 bit | 96 bit       | 3 rounds<br>76.31 ms  | 2 rounds<br>0.12 ms |
| Our HPAKE             | 3 exp + 2 mexp<br>3.48 ms | 3 exp + $t$ mexp<br>17.19 ms | 0 exp<br>0.00 ms | 0 bit        | 5568 bit | 96 bit       | 2 rounds<br>50.48 ms  | 2 rounds<br>0.12 ms |

<sup>1</sup> exp: number of exponentiation; mexp: number of multi-exponentiation.

<sup>2</sup>  $t$  is the sweetword number for each account. For the real running, we set  $t = 20$  as recommended as in [15].

<sup>3</sup> The client stores nothing, but the user needs to memorize the password.

<sup>4</sup> For a fair comparison, we only count the cost of cryptography suites in honeyword mechanism (TLS).

### B. Environment of the deployment

To evaluate the efficiency of our HPAKE, we deploy it in the real-world environment as follows.

- 1) The authentication server  $S$  is deployed in a Docker container running on a remote server in Alibaba Cloud. The Docker container is assigned with 2 CPUs (Intel Core i5-7300HQ CPU @ 2.50Hz 2.50Hz) and 2.0GB memory. In the container, Ubuntu 18.04 with MySQL 7.4.8 is deployed. To allow any client  $C$  from Internet to access  $S$ , we set this container to allow any connection from the Internet.
- 2) The honeychecker  $HC$  is deployed in another Docker container on the same server in Alibaba Cloud. This container is assigned with 1 CPU (Intel Core i5-7300HQ CPU @ 2.50Hz 2.50Hz) and 1.0GB memory. To avoid the same vulnerability affecting the authentication server and honeychecker, we leverage another operating system - CentOS 8.0 - for the honeychecker. Since  $HC$  only allows the connection from  $S$ , this container is set to only allow the connection from local host.

- 3) The client  $C$  is deployed in a laptop computer equipped with an Intel Core i5-7300HQ CPU, 16.0GB memory.
- 4) The communication between  $S$  and  $HC$  is via a local channel. Its round-trip delay time is only 0.123 ms and it has 0.0 packet loss.
- 5) The communication between  $S$  and  $C$  is over Internet. The round-trip delay time of this channel is around 48.693 ms, and it has 0.5% packet loss. The instability of the channel leads to the bulges in Figure 9.

### C. Performance

Here, we only consider the time cost of the exponentiation and multi-exponentiation operations, since the time cost of other operations (e.g., honey encryption and hash) is much smaller. Please note that with Shamir's trick, one multi-exponentiation only needs  $\frac{1}{6}$  more time cost than one exponentiation. In the statistic on real times, we set 20 as the sweetword number  $t$ , since the value is recommended in [15] and large enough for normal accounts (see the discussion in Section III-C).

**Computational cost.** Compared with OPAQUE, our HPAKE only does one more multi-exponentiation on the client (for verifying the signature),  $t - 1$  more multi-exponentiations on the authentication server (for calculating the  $t - 1$  honey session keys). Note the  $t$  session keys can be parallel calculated. Further, considering the computing power of the authentication server, this cost is acceptable for the authentication server. As shown in Figure 8 and Table II, when  $t = 20$ , our HPAKE only costs 3.48 ms on the client, 17.19 ms on the server, and 0.00 ms on the honeychecker.

**Communication cost.** Compared with OPAQUE, our HPAKE only increases two rounds of communication between the server and honeychecker. Since the round-trip delay time (0.12 ms) on the server-honeychecker channel is very small (the server and the honeychecker is deployed on the same server but different Docker container), this increased time is negligible. Compared with honeyword mechanism (for password-over-TLS), our HPAKE decreases the communication rounds between the server and client from 5 to 2. Since the round-trip delay time on the client-server communication is much higher than other parts (including the cost on computation and on the server-honeychecker communication), our HPAKE achieves a great improvement on efficiency than honeyword mechanism. As shown in Figure 9 and Table II, our HPAKE only costs 50.48 ms on the client-server communication and 0.12 ms on the server-honeychecker communication.

**The sweetword number's influence on performance.** As discussed in Section III-C, different accounts may be set with different  $t$  according to their importance. For HPAKE, more honeywords, more cost. Fortunately, we keep the *marginal cost* of adding a honeyword on a small number: 0 for the communication cost; only one multi-exponentiation (about 0.74 ms) on the server's computation cost and one public key (256 bits) on the server's storage cost. If we magnify the sweetword number by a factor of 10 (i.e., setting  $t = 200$ ), we still have an acceptable cost: 150.39 ms and 51,648 bits (6.3 MB) on the server side (the other costs do not change).

**Summary.** Our protocol only costs 71.27 ms for one complete run ( $t = 20$ ), within 20.67 ms on computation and 50.6 ms on communication. This means our design is secure and practical for real-world applications.

## VI. CONCLUSION

We propose the notion of HPAKE, which is the first of its type, achieving the advantages of the honeyword and aPAKE techniques, i.e., detecting the password leakage caused by external attackers and preventing the insider from getting the password plaintext. Using OPAQUE, honeyword mechanism, and honey encryption, we build a concrete HPAKE construction. To analyze the security of our design, we propose a game-based security model and formally prove the security of our design in this model. We implement and deploy the proposed scheme in the real-world environment. The experimental results show that our design is efficient for the real-world applications.

## REFERENCES

- [1] J. Bonneau, C. Herley, P. C. Oorschot, and F. Stajano, "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes," in *Proc. IEEE S&P 2012*, pp. 553–567.
- [2] N. Huaman, S. Amft, M. Oltrogge, Y. Acar, and S. Fahl, "They would do better if they worked together: The case of interaction problems between password managers and websites," in *Proc. IEEE S&P 2021*, pp. 1367–1381.
- [3] D. Pasquini, A. Gangwal, G. Ateniese, M. Bernaschi, and M. Conti, "Improving password guessing via representation learning," in *Proc. IEEE S&P 2021*, pp. 265–282.
- [4] W. Li and J. Zeng, "Leet usage and its effect on password security," *IEEE Trans. Inform. Foren. Secur.*, vol. 16, pp. 2130–2143, 2021.
- [5] "Have i been pwned?" [Online]. Available: <https://haveibeenpwned.com>
- [6] "Yahoo! data breaches." [Online]. Available: [https://en.wikipedia.org/wiki/Yahoo!\\_data\\_breaches](https://en.wikipedia.org/wiki/Yahoo!_data_breaches)
- [7] "Yahoo tries to settle 3-billion-account data breach with \$118 million payout." [Online]. Available: <https://arstechnica.com/tech-policy/2019/04/yahoo-tries-to-settle-3-billion-account-data-breach-with-118-million-payout/>
- [8] Z. Whittaker, "Github says bug exposed some plaintext passwords," <https://www.zdnet.com/article/github-says-bug-exposed-account-passwords/>, 2018.
- [9] S. M. Bellovin and M. Merritt, "Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise," in *Proc. ACM CCS 1993*, pp. 244–250.
- [10] V. Boyko, P. MacKenzie, and S. Patel, "Provably secure password-authenticated key exchange using diffie-hellman," in *Proc. EUROCRYPT 2000*. Springer, pp. 156–171.
- [11] C. Gentry, P. MacKenzie, and Z. Ramzan, "A method for making password-based key exchange resilient to server compromise," in *Proc. CRYPTO 2006*. Springer, pp. 142–159.
- [12] S. Jarecki, H. Krawczyk, and J. Xu, "OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks," in *Proc. EUROCRYPT 2018*. Springer, pp. 456–486.
- [13] M. Abdalla, M. Barbosa, T. Bradley, S. Jarecki, J. Katz, and J. Xu, "Universally composable relaxed password authenticated key exchange," in *Proc. CRYPTO 2020*. Springer, pp. 278–307.
- [14] S. Smyshlyaev, N. Sullivan, and A. Melnikov, "[cfrg] results of the PAKE selection process," 2020. [Online]. Available: [https://mailarchive.ietf.org/arch/msg/cfrg/LKbwodpa5yXo6VuNDU66vt\\_Aca8/](https://mailarchive.ietf.org/arch/msg/cfrg/LKbwodpa5yXo6VuNDU66vt_Aca8/)
- [15] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *Proc. ACM CCS 2013*, pp. 145–160.
- [16] D. Wang, H. Cheng, P. Wang, J. Yan, and X. Huang, "A security analysis of honeywords," in *Proc. NDSS 2018*, pp. 1–18.
- [17] Akshima, D. Chang, A. Goel, S. Mishra, and S. K. Sanadhya, "Generation of secure and reliable honeywords, preventing false detection," *IEEE Trans. Depend. Secur. Comput.*, vol. 16, no. 5, pp. 757–769, 2019.
- [18] K. C. Wang and M. K. Reiter, "Using amnesia to detect credential database breaches," in *Proc. USENIX Security 2021*, pp. 839–855.
- [19] "Passwordless authentication — duo security." [Online]. Available: <https://duo.com/solutions/passwordless>
- [20] W. Li, H. Cheng, P. Wang, and K. Liang, "Practical threshold multi-factor authentication," *IEEE Trans. Inform. Foren. Secur.*, vol. 16, pp. 3573–3588, 2021.
- [21] J. Zhang, H. Zhong, J. Cui, Y. Xu, and L. Liu, "Smaka: Secure many-to-many authentication and key agreement scheme for vehicular networks," *IEEE Trans. Inform. Foren. Secur.*, vol. 16, pp. 1810–1824, 2021.
- [22] J. Srinivas, A. K. Das, M. Wazid, and N. Kumar, "Anonymous lightweight chaotic map-based authenticated key agreement protocol for industrial internet of things," *IEEE Trans. Depend. Secur. Comput.*, vol. 17, no. 06, pp. 1133–1146, 2020.
- [23] S. Jangirala, A. K. Das, N. Kumar, and J. J. Rodrigues, "Cloud centric authentication for wearable healthcare monitoring system," *IEEE Trans. Depend. Secur. Comput.*, vol. 17, no. 05, pp. 942–956, 2020.
- [24] J. Ma, W. Yang, M. Luo, and N. Li, "A study of probabilistic password models," in *Proc. IEEE S&P 2014*, pp. 538–552.
- [25] A. Juels and T. Ristenpart, "Honey encryption: Security beyond the brute-force bound," in *Proc. EUROCRYPT 2014*. Springer, pp. 293–310.
- [26] "Github - cfrg/draft-irtf-cfrg-opaque/ the opaque asymmetric pake protocol." [Online]. Available: <https://github.com/cfrg/draft-irtf-cfrg-opaque>
- [27] S. M. Bellovin and M. Merritt, "Encrypted key exchange: Password-based protocols secure against dictionary attacks," in *Proc. IEEE S&P 1992*, pp. 72–84.

- [28] S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu, “Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online),” in *Proc. EuroS&P 2016*, pp. 276–291.
- [29] H. Krawczyk, “Hmqv: A high-performance secure diffie-hellman protocol,” in *Proc. CRYPTO 2005*, pp. 546–566.
- [30] T. Wu, “The secure remote password protocol,” in *Proc. NDSS 1998*, vol. 98. Citeseer, 1998, pp. 97–111.
- [31] D. Taylor, T. Wu, N. Mavrogiannopoulos, and T. Perrin, “Using the secure remote password (srp) protocol for tls authentication,” *Request for Comments*, vol. 5054, 2007.
- [32] H. Cheng, Z. Zheng, W. Li, P. Wang, and C.-H. Chu, “Probability model transforming encoders against encoding attacks,” in *Proc. USENIX Security 2019*, pp. 1573–1590.
- [33] H. Cheng, W. Li, P. Wang, C.-H. Chu, and K. Liang, “Incrementally updateable honey password vaults,” in *Proc. USENIX Security 2021*, pp. 857–874.
- [34] J. Becerra, P. B. Rønne, P. Y. Ryan, and P. Sala, “Honeypakes,” in *Cambridge International Workshop on Security Protocols*. Springer, 2018, pp. 63–77.
- [35] M. Bellare and P. Rogaway, “Entity authentication and key distribution,” in *Proc. CRYPTO 1993*. Springer, pp. 232–249.
- [36] M. Bellare, W. D. Pointcheval, and P. Rogaway, “Authenticated key exchange secure against dictionary attacks,” in *Proc. EUROCRYPT 2000*. Springer, pp. 139–155.
- [37] R. Canetti and H. Krawczyk, “Universally composable notions of key exchange and secure channels,” in *Proc. EUROCRYPT 2002*, pp. 337–351.
- [38] D. A. Basin, A. Lochbihler, and S. R. Sefidgar, “Crypthol: Game-based proofs in higher-order logic,” *J. Cryptol.*, vol. 33, no. 2, pp. 494–566, 2020.
- [39] R. Canetti, “Universally composable security: A new paradigm for cryptographic protocols,” in *Proc. IEEE FOCS 2001*, pp. 136–145.
- [40] Y. Xiong, C. Su, W. Huang, F. Miao, W. Wang, and H. Ouyang, “Smartverif: Push the limit of automation capability of verifying security protocols by dynamic strategies,” in *Proc. USENIX Security 2020*, pp. 253–270.
- [41] D. Shukla and V. V. Phoha, “Stealing passwords by observing hands movement,” *IEEE Trans. Inform. Foren. Secur.*, vol. 14, no. 12, pp. 3086–3101, 2019.
- [42] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek, “Password cracking using probabilistic context-free grammars,” in *Proc. IEEE S&P 2009*, pp. 391–405.
- [43] “Github - bytemare/opaque/ go implementation of the opaque asymmetric password-authenticated key exchange protocol.” [Online]. Available: <https://github.com/bytemare/opaque/>
- [44] “Github - novifinancial/opaque-ke/ an implementation of the opaque password-authenticated key exchange protocol.” [Online]. Available: <https://github.com/novifinancial/opaque-ke>

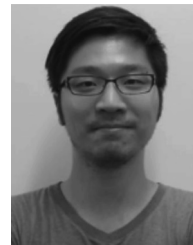


**Wenting Li** received her Ph.D. degree in School of Software and Microelectronics from Peking University in 2021. Now she is currently a lecturer at Peking University, Beijing, China. She has authored over 10 papers in journals or proceedings such as IEEE TII and USENIX Security. Her research interests include authentication protocol and password security.



security and distributed computing.

**Ping Wang** received his Ph.D. degree in Computer Science from the University of Massachusetts, USA in 1996. He is currently a professor with National Engineering Research Center for Software Engineering, and School of Software and Microelectronics at Peking University, China. He is the Director of intelligent Computing and Sensing Laboratory (iCSL), Peking University. Prof. Wang has authored over 100 papers in journals or proceedings such as ACM CCS, USENIX Security, NDSS, IEEE TDSC and IEEE ICWS. His research interests include information



**Kaitai Liang** received the Ph.D. degree in computer science (applied cryptography direction) from the City University of Hong Kong. He is currently an Assistant Professor in Department of Intelligent Systems, Delft University of Technology, The Netherlands. His current research interests include applied cryptography, data security, user privacy, cybersecurity, blockchain security, and privacy-enhancing technology.