Building and Studying a Password Store that Perfectly Hides Passwords from Itself

Maliheh Shirvanian, Nitesh Saxena, Stanislaw Jarecki, and Hugo Krawczyk

Abstract—We introduce a novel approach to password management, called SPHINX, which remains secure even when the password manager itself has been compromised. In SPHINX, the information stored on the device is theoretically independent of the user's master password. Moreover, an attacker with full control of the device, even at the time the user interacts with it, learns nothing about the master password – the password is not entered into the device in plaintext form or in any other way that may leak information on it. Unlike existing managers, SPHINX produces strictly high-entropy passwords and makes it compulsory for the users to register these passwords with the web services, which defeats online guessing attacks and offline dictionary attack upon service compromise. We present the design, implementation and performance evaluation of SPHINX, offering prototype browser plugins, smartphone apps and transparent device-client communication. We further provide a comparative analytical evaluation of SPHINX with other password managers based on a formal framework consisting of security, usability, and deployability metrics.

Index Terms—Authentication, password, password manager

1 Introduction

The central role of passwords for authentication and for gaining access to resources, from casual website visits to national security, is well known. Equally well known are the major security vulnerabilities of such mechanisms spawned by the limitations of human memory and the consequent low entropy of passwords (e.g., [17], [20], [38], [44]). Such low-entropy passwords are vulnerable to both online guessing and offline dictionary attacks that build on password dictionaries from which a significant portion of passwords are chosen. Candidate passwords for authenticating a user to a server can be tested by an attacker through online interactions with the server. Even more seriously, an attacker breaking into a server can mount an offline attack that uses information stored on the server to test the different passwords in the dictionary. Such offline dictionary attacks are an increasingly important concern (see, e.g. [14], [15]), especially in light of frequent attacks against major commercial vendors, as recently experienced, e.g., by PayPal [2], LinkedIn [8], Blizzard [3] and Gmail [7]. The offline attacks are particularly devastating as a single server break-in may lead to extraordinary numbers of compromised passwords [6], [16]. Furthermore, since many users reuse their passwords across multiple services, compromising one service often compromises user accounts at other services.

Numerous approaches have been proposed by researchers and practitioners to improve the security of passwords from the client-side or user-side alone (i.e., without making any changes to a persistent server that uses traditional password-based authentication). One broad class of such approaches is referred to as *password managers* and forms the central focus of this paper.

Traditional password managers (e.g., [1], [12]) allow the user to store and retrieve (*usually* high-entropy) passwords, denoted by rwd, 1for her multiple password-protected services by interacting with a "device" serving the role of the manager (a smartphone or an online third-party service)[1] on the basis of a single (low-entropy) master password, denoted pwd. These rwd's are usually stored on the device encrypted under pwd. In case of online password managers, the user provides pwd to the service, which then unlocks and sends rwd to the user (over a protected channel). In case of smartphone managers, the user enters pwd directly on the device, the device unlocks and displays rwd, and the user then manually copies pwd over to the client machine.

These password managers clearly alleviate the memorization burden on the user, and work well to defeat offline dictionary attacks upon web service compromise, assuming the use of high-entropy rwd's is enforced. However, they are vulnerable to leakage of rwd's in the event the device is compromised or is itself malicious, due to: (1) the storing of the passwords rwd's encrypted under pwd, and/or (2) the need to input the master password to the device (smartphone managers). In the first case, after retrieving encrypted rwd's from the storage unit of the password manager, the attacker can launch an offline dictionary attack against pwd.[2] Such attacks are a serious

- *M. Shirvanian and N. Saxena are with the University of Alabama at Birmingham, Birmingham, AL 35294. E-mail: {maliheh, saxena}@uab.edu.*
- *S. Jarecki is with the University of California, Irvine, CA 92697. E-mail: stasio@ics.uci.edu.*
- *H. Krawczy is with IBM Research, New York, NY 10598. E-mail: hugo@ee.technion.ac.il.*

Manuscript received 5 Feb. 2019, accepted 22 Feb. 2019, Date of publication 14 Mar. 2019; date of current version 30 Aug. 2019.
(Corresponding author: Maliheh Shirvanian.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TDSC.2019.2902551

1. Password managers that store rwd's on the client machine (browser) itself (e.g., [11]), make it very hard for the user to move from one client to another, and may actually be equivalent to online managers to enable syncing of stored passwords across multiple clients through an online server.

2. If *all* rwd's are *fully* random then with careful enough encryption an offline dictionary attack in this case may be avoided; yet concern (2) would remain in this case too.

concern in light of recent breaches against commercial online managers [43] and exfiltration approaches discussed in the literature [35]. In the second case, pwd is directly exposed to the attacker. With the advent of mobile computing, malware that can compromise mobile devices is becoming a major threat [4], [10], [46], and thus existing smartphone managers open up a significant vulnerability of exposing pwd upon entry and/or leaking rwd's upon offline dictionary attack.

Cracking-resistant password encoding strategies have been proposed in the literature to render offline dictionary attacks ineffective [22]. They introduce the notion of outputting decoy passwords to an attacker who compromises the manager and attempts to decrypt the passwords with a wrong master password. Since the attacker is not aware of the correct password, any attempt to login with the decoy passwords can be prevented on the server, and raise an alert. However, such a scheme seems to be vulnerable to an attack presented in a very recent work [28], based on differences in the distribution of the passwords.

Other advanced password management solutions have been proposed that do not require storage of rwd's [29], [39], [45]. For example, PwdHash [39], maps pwd to a rwd by hashing (pwd, domain) pair and registering it as a strong password with the server. PwdHash deterministically transforms a user's password into a more complex password but this transformation does not protect against offline dictionary attacks at a compromised web service. Moreover, if a user uses the same memorable password pwd with PwdHash for different services, the compromise of a single server leads to the discovery of pwd via an offline dictionary attack and then to the (deterministic) calculation of all the user's passwords derived from pwd. In our case, the compromise of a server does not help the attacker learn either the randomized password rwd used for that server or the underlying password pwd. We provide a brief review and analysis of several password managers in Section 4.

In this paper, we introduce, build and study SPHINX, a new password manager that offers a high level of security even in case the password manager itself is compromised. SPHINX's most appealing features are: (1) the information stored in the device is *information theoretically independent* of the user's master password pwd; hence, an attacker breaking into the device learns *no information* on pwd or the user's individual passwords rwd's; and (2) an attacker with full control of the device, even at the time the user interacts with it, learns *nothing* about pwd; pwd is never entered into the device in plaintext form or in any other way that may leak information. The above properties hold unconditionally, even against a computationally unbounded attacker.

Moreover, SPHINX produces strictly high-entropy rwd's and enforces the users to register these passwords with the web services, while current password managers may let the users choose and register low-entropy passwords thereby still opening up the vulnerability to online guessing attacks and offline dictionary attacks upon web service compromise. As an added advantage over existing managers that require some form of secure channels between the device and the client machine, SPHINX can work with *non-confidential channels* offering additional layer of security. Given the numerous vulnerabilities of PKI to certification failures and man-in-the-middle (MitM) attacks (either due to programmatic errors or human mistakes), e.g., [24], [27], [42], relying upon secure channels in existing online managers may be problematic, a situation SPHINX avoids.

The design and security of SPHINX is based on the device-enhanced password authenticated key exchange (DE-PAKE) model of Jarecki et al. [31] that provides the theoretical basis for this construction and is backed by cryptographic proofs of security. The core technique is the use of an efficient oblivious pseudo random function (OPRF) scheme [25], [26] that transforms a human-memorable password into a random password with the aid of a device without the need to store the passwords on the device and without the device learning anything about the password even when computing on it. Specifically, when using SPHINX, for each service with which the user has an account, the device stores a unique key k. This key is used to map the user-memorized password pwd (input by the user into the client machine) into a randomized password $\mathsf{rwd} = F_k(\mathsf{pwd}|\mathsf{domain})$ using the (oblivious) PRF F_k based on a simple protocol between the device and the client.

In sum, SPHINX offers the following *simultaneous* combination of security features:

1) Resistance to online guessing attacks, due to the use of high-entropy and mutually independent passwords rwd registered with web services.
2) Resistance to offline dictionary attacks under server compromise (or server being malicious), due to the server storing a salted one-way hash of rwd, a randomized input [31].
3) Resistance to phishing attacks, due to the use of website domain in the computation of rwd.
4) Resistance to offline dictionary attacks under device compromise (or device being malicious), in particular hiding the user's master password information theoretically.
5) Resistance to eavesdropping and man-in-the-middle attacks on the device-client channel *without* the need to establish a confidential channel (the properties of the OPRF protocol executed over this channel ensure that no additional protection is needed).

The last two security properties are unique to SPHINX, not offered by any existing password managers. The first security property is provided by default in SPHINX, while this property may or may not be provided by existing managers depending upon whether or not high-entropy passwords are enforced by the manager. The contrasts are summarized in Table 1. SPHINX also increases security against online guessing attacks by making use of a strictly random password registered with the service.

Security is not the only attractive feature of SPHINX. SPHINX also strives to provide a level of user experience close to that of password-only authentication but without the burden of remembering multiple passwords, and with full-entropy per-site passwords.

Detailed Contributions. While SPHINX is suitable for different types of devices, here, we report on its concrete instantiation developed on smartphones given their popularity and trustworthiness as password managers as suggested in the past [32].

1) *A Novel Password Manager* (Section 2): We introduce SPHINX, a novel cryptographic password manager

TABLE 1
Security Properties of SPHINX in Contrast to Current Managers

Resistance to:	SPHINX	Current Managers
Offline dictionary attacks under server compromise (or server being malicious)	Yes, by enforcing random independent passwords per site.	Only if manager enforces high-entropy per-site passwords.
Phishing attacks	Yes, by incorporating domain name.	Only the hash-based managers [29], [39], [45].
Offline dictionary attacks under device compromise (or device being malicious)	Yes, *perfect security* (information theoretic secrecy).	No, passwords are stored or entered on the manager.
Eavesdropping and/or man-in-the-middle attacks on the device-client channel	Yes, without the need to establish a confidential channel.	Enforced by external mechanisms or physical security assumptions*.

* *Current online managers require confidential and authenticated channels, while current smartphone managers require confidential channels.*
Highlighted cells represent the unique advantages offered by SPHINX compared to other managers.

application that *perfectly hides passwords from itself.* SPHINX is a novel application of the general device-enhanced password key exchange (DE-PAKE) framework fron [31]. DE-PAKE is a broad modular cryptographic primitive with several possible applications. SPHINX is one such important applications (not studied in [31]). using an instantiation of DE-PAKE that works *with no modification*on the current web services that use password-only authentication (in particular, allowing for the use of SPHINX with typical TLS-enabled services that used the password-over-TLS protocol). This practical application was not studied by the authors of [31].

2) *System Design and Implementation* (Section 3). We present the design, implementation and performance evaluation of a full smartphone-based SPHINX system offering a prototype browser (Chrome) plugin and a device (Android) app. As a main component of our design, we highlight and address the challenges associated in building transparent and robust bidirectional browser-device communication.
3) *Comparative Analytical Exposition* (Section 4). We provide a comprehensive analytical evaluation of SPHINX comparative to other password management approaches based on an adaptation of an existing elaborate framework of security, usability and deployability metrics [21]. Our analysis shows that SPHINX is a preferred choice assuming the presence of a device during the login process.

Scope of the Work. Just like any other smartphone password manager (and even currently deployed two-factor authentication mechanisms), our studied SPHINX instantiation assumes the availability of the smartphone during the authentication process. Based on this assumption, it simultaneously provides security properties 1-4 above, and a satisfactory level of user experience when compared to regular password-only authentication.

Like *any* password management, providing protection in the event of client compromise is beyond the scope of SPHINX. While some protection is provided by SPHINX in the form of anti-phishing defenses and the ability to block remote adversarial activity by requiring user confirmation on the device upon SPHINX invocation, comprehensive defenses based on two-factor authentication (TFA) techniques, entitle server-side changes which password managers avoid. Integrating SPHINX with a TFA solution is possible but we leave this as an item of future work.

A further extension of our work could support a SPHINX online instantiation that replaces or complements the smartphone-based instantiation (e.g., as a main password manager or as a backup option to the smartphone-based instantiation). Fortunately, the security properties of SPHINX, particularly its resistance to device compromise and security against active man-in-the-middle attackers on client-device channel without the need for confidential channels, make the online variant very appealing. Although we discuss how our work can be extended in the future to such an online case, building and testing the full implementation is beyond the scope of the current paper.

Contribution over Conference Publication. This paper is an extension of our ICDCS 2017 paper [40]. In this submission, we extend our ICDCS 2017 paper by presenting a comprehensive analytical comparison of SPHINX with other password managers based on a formal framework consisting of security, usability and deployability metrics. We also provide a simple inspection analysis comparing the user task flows in SPHINX, password-only authentication and other device-based password managers.

2 Our Approach

2.1 Background

We first review the notion of Device-Enhanced Password-Authenticated Key Exchange (DE-PAKE) introduced in [31], a cryptographic primitive which gives rise to our SPHINX application. DE-PAKE securely transforms a user-memorable password into a full-entropy random string (strong password) by leveraging a secondary device, and then uses this random password as an input to any password-based authenticated key exchange protocol (PAKE) [18]. In [31], authors developed such a "password-to-random" (PTR) protocol functionality and studied its composition with any PAKE protocol, giving rise to a DE-PAKE protocol that is resistant to online guessing and offline dictionary attacks (under server and device compromise), without the need for confidential communication between device and the client.

In DE-PAKE protocol model, there are four parties: user, client, server, and device, individually denoted U, C, S, and D, respectively. U's goal is to authenticate itself to S via C by making use of a simple password and a personal device

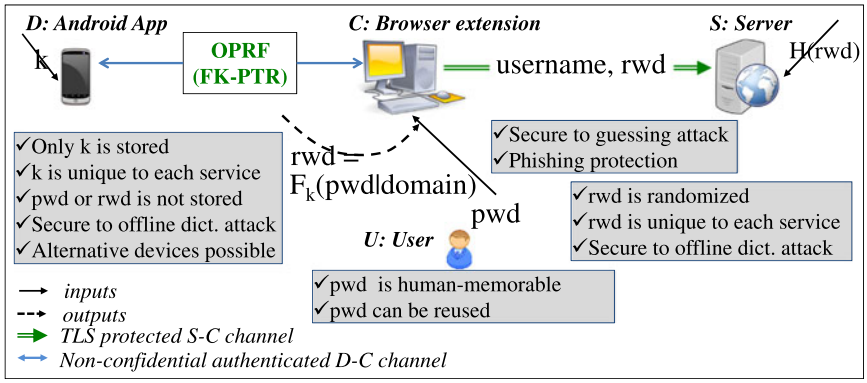

Fig. 1. A high-level overview of SPHINX. U enters memorable password pwd and approves the communication on D (explicit consent), D and C run an OPRF protocol (instantiated as FK-PTR) to construct a randomized password rwd, C sends rwd to S over SSL/TLS to authenticate to the service. A smartphone instantiation is developed and tested in the paper. However, the phone can be replaced with an online service.

D. The protocol has two phases: initialization and authenticated key exchange. In the initialization phase, U chooses a password from a given dictionary Dict. The initialization phase also includes the device-client communication that establishes the state stored at D, as well as interaction with S producing a user's state $\sigma_S(U)$ that S stores while U only remembers its password. After initialization, the link between U and D is subject to the same MitM adversarial activity as in the links between U and S. In the authenticated key exchange phase, U interacts with D, C and S over adversary-controlled channels to authenticate itself to S and establish session keys to protect communication with S.

The PTR protocol design [31] follows the "password hardening" approach of Ford and Kaliski [25] (hence termed FK-PTR). DE-PAKE model assumes a fully capable man-in-the-middle attacker active on all the links between all parties and one is allowed to compromise servers and devices at will. The idea in the FK scheme is for C to interact with one or more auxiliary servers to map the user's (non-random) password into a (random) secret (the "hardened password"), taken from a dictionary unknown to the attacker. This secret is then used by U to authenticate to S. DE-PAKE adopts this idea but replaces the auxiliary servers with a single device, where the device-client channel is not *confidential*.

2.2 SPHINX Overview and Features

SPHINX is a compelling application of DE-PAKE [31]. It is a password manager (Fig. 1), transparent to any existing service that deploys password authentication in which, U registers a hardened randomized password rwd with S, but only remembers a memorable password pwd that could be the same for multiple accounts (we use the terms master password and memorable password interchangeably). For each server S with which the user U has an account, the device D stores a unique key k. The key is used to map the pwd into a randomized password $\mathsf{rwd} = F_k(\mathsf{pwd}|\mathsf{domain})$ using the (oblivious) PRF F_k. pwd and rwd are never stored in C and (in contrast to the current managers 4) neither rwd nor pwd is ever stored in or exposed to D. Instead, D and C run the PTR protocol to *obliviously* compute rwd at the login time. SPHINX offers several key security guarantees simultaneously (individually as well as combined), namely:

1) *Resistance to online guessing attacks*: SPHINX increases security against online guessing attacks by making use of a strictly random password registered with the service.
2) *Resistance to offline dictionary attacks under server compromise (or server being malicious)*: In SPHINX, the server stores the salted-hash of the randomized password rwd, and hence the compromise of a server does not help the attacker to learn rwd (or pwd). Note that a dictionary attack is *infeasible* since rwd does not belong to a dictionary known to the attacker.
3) *Resistance to phishing attacks*: SPHINX combines pwd with the domain name of the web service to compute rwd to provide protection under a phishing attack. The phisher can not even perform an offline dictionary attack to learn rwd or pwd.
4) *Resistance to offline dictionary attacks under device compromise (or device being malicious)*: SPHINX does not leak any information about pwd or rwd to the device or a potential malicious code running on the device, since neither the pwd nor rwd is stored or entered on the device. Therefore, any offline attack against the device (remote or physical) does not reveal any information about the pwd or rwd, even during a user's active session.
5) *Resistance to eavesdropping and man-in-the-middle attacks on the device-client channel*: SPHINX does not require a confidential D-C channel, but is secure against eavesdropping and MITM attack over D-C channel.

Security is not the only compelling feature of SPHINX. It also offers the following usability advantages:

1) *Use of a human-memorable password*: In SPHINX, the user simply remembers pwd but registers a strong randomized password rwd with the service. Moreover, pwd can be reused over multiple accounts without compromising security, while the randomized password rwd is unique to each service.
2) *Easy password updates:* Rather than asking the user to frequently change the password and memorize the updated password (as is a common practice today), only the key on the device can be changed (e.g., requesting key update for a service through SPHINX extension). Therefore, if a given service requires frequent password updates, the user needs to update only the key for that specific account while still

Setup

- *Group G.* The scheme works over a cyclic group G of prime order q, $|q| = \tau$, with generator g.
- *Hash functions H, H'* map arbitrary-length strings into elements of $\{0,1\}^\tau$ and G, respectively, where τ is a security parameter.
- *OPRF.* For a key $k \leftarrow Z_q$, we define function F_k as $F_k(x) = H(x, (H'(x))^k)$.
- *Parties.* User U, Client C, Device D, Server S.

Initialization Phase

- D chooses and stores OPRF key $k \leftarrow Z_q$;
- U chooses and remembers a memorable password pwd $\leftarrow$ Dict;
- C and D interact to construct "randomized password" rwd $= F_k(\mathsf{pwd}|\mathsf{domain})$ on input pwd from U and k from D;
- Server S stores one-way hash of the "randomized password" rwd.

Login Phase

- **Client-Device Interaction (FK-PTR)**
  1) C chooses $\rho \leftarrow Z_q$; sends $\alpha = (H'(\mathsf{pwd}|\mathsf{domain}))^\rho$ to D.
  2) D checks that the received $\alpha \in G$ and if so it responds with $\beta = \alpha^k$.
  3) C sets rwd $= H(\mathsf{pwd}|\mathsf{domain}, \beta^{1/\rho})$.
- **Client-Server Interaction**

  U authenticates to S using the "randomized password" rwd submitted over SSL/TLS channel.

Fig. 2. SPHINX protocol details.

continuing to use the same original master password. This feature provides an improved level of usability.

3) *Use of multiple types of devices:* Since no information about pwd or rwd is learned by the device and the device-client channel does not need to be a secure channel, a personal device could be a smartphone, a wearable device (e.g., smartwatch) or even an online service (as will be discussed in Section 5).

All the features are offered by SPHINX simultaneously, while other password managers only provide a partial subset of these properties, especially no other scheme provides the last two properties, as will be elaborated in Section 4.

2.3 SPHINX Protocol and System Details

In the standard password-only authentication schemes, U authenticates to S using pwd. In the SPHINX protocol as shown in Fig. 2, C runs an instance of password authentication protocol not on pwd, but on value rwd $= F_k(\mathsf{pwd}|\mathsf{domain})$ where F is a pseudorandom function (PRF) and k is a key held by D. Before authenticating to S, C contacts D (through a client application) and obtains the rwd value using a special-purpose (oblivious) PRF-evaluation protocol. Without knowledge of k, the value rwd has full entropy in the range set of function F and hence dictionary attacks do not apply against rwd (not even if the server is compromised). Since D holds only the PRF key k and S holds only information related to pseudorandom value rwd $= F_k(\mathsf{pwd}|\mathsf{domain})$, we can ensure that offline attacks against pwd are also infeasible when S or D are compromised. In the case of D compromise, the user's password is not exposed. Rather, security reduces to the security of the memorable password (the attacker who obtains the device can attempt to guess the memorable password only in an online attack against S).

SPHINX PRF protocol is defined in [31] as *PTR*, and is designed in a way that neither the device or a MITM learn anything about the password (pwd or rwd) and no one other than the device learns anything about the key k.

The implementation of the SPHINX protocol is based on an instantiation of the DE-PAKE primitive [31]. This instantiation assumes a cyclic group G of prime order q, $|q| = \tau$, with generator g. At initialization, U chooses and remembers pwd while D chooses and stores $k \leftarrow Z_q$. To retrieve rwd, C first blinds pwd by raising the hashed value $H'(\mathsf{pwd}|\mathsf{domain})$ to a random exponent ρ, and sends it to D. This perfectly hides pwd from D and from any eavesdropper on the C $-$ D link. D checks that the received value is in the group G and if so it raises it to the secret exponent k. Now, C can de-blind this value by raising it to the power $1/\rho$ to obtain $H'(\mathsf{pwd}|\mathsf{domain})^k$. Finally, C hashes this value with pwd to obtain rwd.

Since rwd is a function of pwd and k, users can update it by either choosing a new pwd or updating k. The latter provides a higher usability since the user does not require to update and remember a new pwd.

Note that D contains no information related to pwd hence an attacker interacting with D or even breaking into it, learns nothing about pwd. Also, C does not run any test on the value reconstructed in the FK-PTR protocol. Hence, an attacker that interacts with C in the role of D does not learn anything about pwd from watching the behavior of C. These "obliviousness" and minimality properties of FK-PTR are essential to achieve PTR security. The security of SPHINX directly follows from the security of PTR and DE-PAKE. For formal security arguments underlying PTR and DE-PAKE, we refer the reader to [31].

3 Design, Implementation & Performance Evaluation

We instantiate the SPHINX system using the smartphone as the device serving the role of the password manager. The resulting SPHINX system has two essential components, namely, the browser extension and the Android application communicating over an authenticated channel.

3.1 SPHINX Browser Extension

Step 1–Reading the Password. The browser extension listens to the keyboard events generated on login-page and gets activated once a predefined "@@" password prefix or the "F2" function key is entered in the password field. This design decision is similar to the approach of PwdHash [39] and allows the user to choose whether to use the service for a particular website or not (i.e., only passwords that are preceded by the prefix undergo the protocol). After getting activated, the extension reads the input password. We note that the additional password prefix is only a design choice and can be discarded from the design at no additional security/usability cost. For example, an alternative design choice is

to ask the user to enable SPHINX for each service at the enrollment time.

Step 2–Hashing the Password into the Elliptic Curve. The entered password is input in a "Hash-into-Elliptic-Curve" function. We call this function H' (Fig. 2). We implemented H' using the Stanford Elliptic Curve Cryptography and Core JavaScript libraries for curve and field computation, and CryptoJS library for SHA-256 computation. The Hash-into-Elliptic-Curve function maps the password into a point on NIST P-256 curve. In this implementation, SHA-256 of the input and the iteration counter is computed and truncated into an element in Z_q, and the computed value is considered as the x coordinate of a point on the curve if the y value associated with it is a quadratic residue (i.e., x and y satisfy the curve equation). Otherwise, the same computation is repeated until a curve element is obtained. Such a point on the curve is the output of the hash function.[3] To add resistance against phishing attacks, password is concatenated with the domain name of the website and then is input into H'.

Step 3–FK-PTR OPRF Protocol. After computing the hash, the extension follows its role in the OPRF function to blind the password. The OPRF function [30] is defined as $F_k(x) = H(x, (H'(x))^k)$ with input x from the client and k from the device. The OPRF works over group G of prime order p, which in our implementation is an elliptic curve NIST P-256 group. The input to the OPRF function is the password concatenated with the domain name of the visited page. As described in Fig. 2, the extension picks a random number $\rho \in Z_q$ and raises hash value of the input to the power ρ (note that the use of the blinding factor ρ hides the password with information-theoretic security), and sends it to the device (we call this value α). In response, the extension receives $\beta = \alpha^k$ from the device. After checking the group membership of β, the extension reconstruct the randomized password by raising the received value to the power of $\rho^{-1} \in Z_q$ and then computing SHA-256 hash of the calculated value. We used Stanford Random Number Generator JavaScript API and CryptoJS SHA-256 to generate the random number ρ. The communication channel between the extension and the device will be discussed in more detail in Section 3.3.

Step 4–Entering the Randomized Password. The output of the OPRF is encoded to a random combination of letters, numbers and symbols matching the password requirement of the visited website as per the encoding functionality suggested in [39], and is re-entered in the password field of the login page.

Extension Option Page. The extension has an option page accessible from the `options` tab under the extension name in `chrome://extensions`. Features included in the options page encompasses "prompting the randomized password" to show rwd once pwd is entered, "increment offset" to communicate with the device to update the key for a specific service, and "device IP address" for WebSocket configuration.

3.2 SPHINX Android Application

Step 1–Starting the FK-PTR Protocol. In the first step the Android app receives $\alpha = H'(pwd|domain)^\rho$ from the client, checks the group membership of α, and computes $\beta = \alpha^k$. The OPRF key k is picked by the device at the initialization phase and is stored on the device. An update for k initiates from the client's browser extension that communicates with the device to set a new key for a service. All elliptic curve functions in our app are based on Java Security and Spongy Castle libraries.

Step 2.1–Explicit Consent Mode. In this optional step, which we consider as the primary design option for SPHINX an alert message is displayed on the device to make the user aware of the ongoing login process. The user is required to confirm the alert before the protocol continues to the next step. We call this alert box the explicit consent on the device. This design choice is made to ensure that the device does not respond to unauthorized requests (without user's awareness), preventing an attacker who obtains the user's master password from authenticating to the service.

Step 2.2–Zero-Interaction Mode. In the second option, user can disable the explicit consent requirement, for the application to run on the device with zero interaction with the user, without seeking for user's approval. This design choice is assumed to be more usable and transparent to the user (we refer to this feature as "Physically Effort-less" in Section 4). However, an attacker who has obtained the master password (e.g., with a shoulder surfing attack), might be able to login to the service (we refer to this feature as "Resilient-to-Physical-Observation" in Section 4).

Step 3–Completing the FK-PTR OPRF Protocol. In this step the device sends β to the client to complete its role in the OPRF.

3.3 Device-Client Authenticated Channel

In our first implementation of the SPHINX system for Google Chrome browser, we used the WebSocket protocol to establish communication between the device and the Chrome Extension. For the client to initiate the WebSocket communication by sending α, the device needs to be set-up as an HTTP server, the client being the HTTP client. We set up an HTTP server on the device using NanoHttpd Java application [9], adapted for Android.

In later implementation of SPHINX, we decided to use Google Cloud Messaging (GCM) to provide a more stable connection between the device and the client. Note that since our application does not require the device-client channel to be secure, trusting on GCM would not at all affect the security of our approach. To our knowledge, this is the first implementation of GCM that makes a bi-directional connection between two typically considered GCM clients (a phone and a browser) without the need for any additional relaying server.

3.4 Performance Evaluation

The overall execution time of the SPHINX and performance of different major tasks on am LG G3 smartphone and the client-side Chrome extension (on MacBook Air laptop with a 1.3 GHz Intel Core i5 processor and 4 GB of memory) is evaluated over 10,000 iterations, and the averaged results are reported in Table 2. The total execution time for both the WebSocket and GCM implementation are shown. We excluded the time of human interaction with the system (manual pwd entry, and explicit consent on the device) from the evaluation. We also excluded authentication to the

3. An alternative, robust to side channels, is to use a hashing-into-the-curve mechanism such as Elligator 2 [19].

TABLE 2
Performance Analysis of the Implemented SPHINX Protocol for NIST P-256 Curve and 128 Bit OPRF Key; * The Total Time Excludes Users' Interaction

	Task	Delay
Device	Group Membership ($\alpha \in G$)	0.36 ms
	Scalar Multiplication ($\beta = \alpha^k$)	71.00 ms
Client	EC-Hash ($H'(\mathsf{pwd}\|domain)$)	2.23 ms
	Scalar Multiplication ($H'(\mathsf{pwd}\|domain)^\rho$)	54.23 ms
	Inverse ($1/\rho$)	0.23 ms
	SHA256 Hash (H)	0.07 ms
	$H(\mathsf{pwd}\|domain, \beta^{1/\rho})$	52.30 ms
Total Time*	Websocket	510.00 ms
	GCM	400.54 ms

service, as this is the same in all schemes. Communication between C and D is timed for a 10 Gbps WiFi Internet.

Based on our evaluation, for all parties, the most costly computation is Elliptic Curve exponentiation (71.00 ms on the extension, and 52.30 ms on the Android app with the mentioned libraries). The overall execution time of SPHINX protocol is around 500 ms and 400 ms, for WebSocket and GCM communication, respectively (excluding human interaction), which seems reasonably efficient.

3.5 User Task Flow Analysis

We analyzed the user task flows in: password-only, SPHINX in Zero-Interaction Mode, SPHINX in Explicit Consent Mode, and currently deployed device-based password managers. Table 3 shows the steps a user follows to authenticate by her username and password to a typical server in four different approaches. Compared to password-only systems, SPHINX in Zero-Interaction Mode does not impose any additional burden to the user, and SPHINX in Explicit Consent Mode requires the confirmation on the device. In contrast, currently deployed device-based password managers require the user to input master password on the device, and then copy the password from the device to the terminal manually. Just based on inspection of the user task flows, it is clear that current device-based password manager will have much lower usability compared to the other three approaches. SPHINX in Zero-Interaction Mode also seems very close to password-only in terms of usability.

4 SPHINX versus Other Managers

We analyze SPHINX with respect to several security, usability and deployability metrics, and compare it with many other managers. Our analysis is summarized in Table 4.

The listed metrics for comparison are built upon the evaluation framework of [21], which was designed to assess web-based authentication schemes. We base our comparison upon a similar set of metrics but refine and extend the list to meet the characteristics of password managers. Our list may be independently used as a specialized framework to evaluate password managers.

The list of studied schemes is comprehensive and includes three main categories of password managers: (1) *Hash-based password managers*, in which the passwords are generated by applying a cryptographic function to the master password and a tag (e.g., the visited website's domain name). In addition to SPHINX, we included, in the comparison, three well-recognized academic works, namely, PwdHash [39], Password-Multiplier [29], and Passpet [45]; (2) *Traditional password managers*, that are usually available as browser extensions, desktop applications, token-based application (e.g., USB key or a smartphone apps), and online services. We included highly ranked or widely used commercial password managers in the list, including Firefox [11], LastPass [12], 1Password [1], Dashlane [5], and RoboForm [13]. We also included an instance of a smartphone password manager, Tapas [37], that is similar in functionality to traditional password managers, but unlike others, runs a protocol between the device and the terminal to encrypt, store and retrieve the password. In this study, we exclude approaches that require service-side changes (e.g., Phool Proof [34], MP-Auth [36], and Pico [41]).

4.1 Analyzed Schemes

We refer to the existing text-based password authentication web services as *password-only* (*PM0*) ("without password manager (PM)"). All the other schemes listed below are compared with this model. Any scheme that can provide similar level of usability as password-only while offering additional security features is preferable.

Hash Based Password Managers

PM1. SPHINX. This refers to our proposed scheme. In our analysis, we consider the primary design of SPHINX in the explicit consent mode, and we will adjust any changes in each of the metrics in case of the zero-interaction mode.

PM2. PwdHash. This refers to a mechanism in which the browser extension computes a deterministic (i.e., unkeyed) hash of input pwd and the domain name of the visited page to construct a randomized password. The domain name is used for resistance against phishing attacks. Although the user can assign different tags to each service (rather than the domain name), in real-life, users will typically choose a weak predictable tag.

PM3. Password Multiplier. Similar to PwdHash, Password Multiplier is a browser extension that applies an unkeyed cryptographic hash function of the master username, master

TABLE 3
Users' Task Flow in Different Password Managers (Usability Level Seems to *Decrease* from *left to right*)

Password-Only	SPHINXZero-Interaction Mode	SPHINXExplicit Consent Mode	Device-based Password managers
1. Enter username	1. Enter username	1. Enter username	1. Enter username
2. Enter password	2. Enter master password (on client)	2. Enter master password (on client)	2. Enter master password (on device)
		3. Click confirm on the device	3. Read and enter the password from device to terminal
	[Password is transferred to the webpage automatically]	[Password is transferred to the webpage automatically]	

TABLE 4
SPHINX versus Other Password Managers

	SECURITY								USABILITY				DEPLOY.	
	S1. Unique-Password-Enforcer	S2. Resilient-to-Phishing	S3. ODA-Resistant-Server-Compromise	S4. Storeless	S5. ODA-Resistant-Device-Compromise	S6. Resilient-Upon-Theft	S7. Resistant-to-MITM-D-to-C	S8. Resilient-to-Physical-Observation	U1. Memorywise-Effortless	U2. Password-Update-Not-Necessary	U3. Scalable-for-Users	U4. Physically-Effortless	D1. Client-Compatible	D2. Nothing-to-Carry
PM0. Password-Only	n	n	n	y	n	y	-	n	y	y	n	y	y	y
PM1. SPHINX														
a) Smartphone Explicit Consent	y±	y	y	y	y	y	y	y	y	n	y	n	n	n
b) Smartphone Zero Interaction	y±	y	y	y	y	y	y	n	y	n	y	y	n	n
c) Online	y±	y	y	y	y	y	y	n	y	n	y	y	n	y
PM2. PwdHash	y	n⊙	n◇	y	n	y	-	n	y	n	y	y	n	y
PM3. Password Multiplier	y	n⊙	n◇	y	n	y	-	n	y	n	y	y	n	y
PM4. Passpet	y	n⊙	n◇	y	n	y	-	n	y	n	y	y	n	y
PM5. Firefox	n	n	n*	n	n	n	-	n	y	y	y	y	y	y
PM6.1. Client-based Managers	n	n	n*	n	n	n	-	n	y	y	y	n	n	y
PM6.2. Token-based Managers	n	n	n*	n	n	n	y†	n	y	y	y	n	n	n
PM6.3. Online Managers	n	n	n*	n	n	n	y‡	n	y	y	y	n	y	y
PM7. Tapas	n	y	n*	n	n	n	y●	y	y	y	y	y	n	n

Notes:
± *Enhanced by the use of two uniqueness parameters, domain name and OPRF key.*
⊙ *They provide higher resistance compared to password-only, but still an ODA is possible after the phishing attack.*
◇ *Unless the user chooses a randomized master password.*
* *Unless the user chooses a randomized password for each account.*
† *Establishment of confidential channel necessary.*
‡ *Establishment of confidential and authenticated channel necessary.*
● *Establishment of confidential and authenticated channel necessary.*
"ODA-Resistance" denotes resistance to offline dictionary attacks. Other metrics are drawn from [21].

password and the target website domain to generate a randomized password for a given user account. It offers resistance to phishing attacks in the same way as SPHINX and PwdHash.

PM4. Passpet. Similar to PwdHash and Password Multiplier, Passpet computes an unkeyed hash of a password to construct a randomized password that will be registered with the service. To provide phishing resistance, Passpet incorporates domain name of the target website in hash function computation (the same way as SPHINX and PwdHash).

Note that of all the above methods, only SPHINX uses a secret keyed function to compute the randomized password.

Traditional Password Managers

PM5. Firefox Password Manager. Firefox Password Manager stores the usernames and the passwords in the terminal storage, and automatically fills the login form when the users visits a website. Firefox Password Manager can work with or without a master password (i.e., with or without encrypting the stored passwords). In our analysis, we consider the version with master password (since this is the fair setting to compare with SPHINX and other password managers). Although we pick Firefox as an instance of desktop password managers, other browsers also offer similar built-in password managers and our comparative analysis applies to them exactly the same way.

PM6. Commercial Password Managers. This category of password managers can further be divided into: *PM6.1. Client-based* (e.g., browser plugin), *PM6.2. Token-based* (e.g., a smartphone app), and *PM6.3. Web-based* (e.g., cloudbased service) commercial password managers, that are often different in the offered security or usability properties. Lastpass, 1Password, Dashlane, and Roboform are instances of commercial cross-browser, cross-platform password managers that store the passwords on some computational device (client/token or web server) and lock them using a master password. They offer several other features such as account syncing, password generation, and form auto fill that are orthogonal to the features of SPHINX and can be adopted.

PM7. Tapas. Tapas is a device-based password manager that suggests a *dual-possession authentication*, leveraging a desktop computer and a smartphone, which runs a protocol between the browser extension (Manager) and the smartphone (Wallet) over a "secure channel" to store the encrypted password on the Wallet and retrieve it when the user visits the webpage. Tapas maintains security of the managed passwords by encrypting and storing the passwords on a smartphone, and keeping the decryption key inside the browser on the paired computer. The dual-possession feature of Tapas helps to protect the user-chosen password in case of offline attacks against the device (unlike other

traditional token-based managers). SPHINX achieves this property without the need to establish a confidential channel, between C and D.

4.2 Metrics and Comparative Analysis

Security Metrics

S1. Unique-Password-Enforcer (captured in S6, Resilient-to-Leaks-from-Other-Verifiers, in [21]): SPHINX, PwdHash, Password Multiplier, and Passpet uniquely select a randomized password for each website. All these mechanisms incorporate domain name of the target website (which is assumed to be unique) in their cryptographic calculation. SPHINX being a *keyed mechanism* assigns different keys to each service, and, even without incorporating the domain, generates a different/unique secure password. In the other managers, it is up to the user to select different passwords for each website. Hence, users may prefer to reuse their password over different services, which is a major security issue.

S2. Resilient-to-Phishing (S7 in [21]). As mentioned earlier, SPHINX, PwdHash, Password Multiplier, and Passpet, incorporate domain name of the webpage in their calculation (precisely in a hash function), and therefore are resistant to phishing attacks. However, unlike SPHINX, other schemes are susceptible to offline dictionary attacks as the phisher can recover the master password, given the hashed password corresponding to the phisher's domain. Tapas provides phishing protection by ensuring that passwords are submitted to the exact site (determined by the site's URL and SSL/TLS certificate) they were registered with. Other managers that we are studying do not provide resistance against phishing.

S3. Offline-Dict-Attack-Resistant-Server-Compromise (captured in S5, Resilient-to-Internal-Observation, in [21]): This feature is *unique* to SPHINX and no other password manager offers security against offline dictionary attacks upon service compromise (unless the user deliberately chooses a random password to register with the service). SPHINX cryptographically randomizes the memorable password, but unlike other hash-based password managers, the cryptographic function is not a deterministic hash function. The cryptographic function is an Oblivious-PRF function in which the browser extension inputs the password (and the domain name for phishing protection) and the device inputs a secret key to reconstruct the password. Therefore, the password that is registered with the service is cryptographically random and attacker cannot perform a dictionary attack against the server. Although, PwdHash, Password Multiplier, and Passpet generate a randomized password, the password is derived from a deterministic hash function on input of a master password (and a predictable domain name), that is susceptible to an offline attack against the master password, upon learning the password hash on the server.

S4. Storeless: We refer to mechanisms that do not store the password encrypted or unencrypted as storeless. Except for the hash-based password managers in our list (i.e., SPHINX, PwdHash, Password Multiplier, and Passpet) that compute the password on the fly, all other mechanism store the password encrypted or unencrypted (on the token, on the client machine, or on an online server). Passpet is the only hash-based password manager that stores domain names, which might expose the privacy of the user, but this does not affect its security.

S5. Offline-Dict-Resistant-Device-Compromise (captured in S5, Resilient-to-Internal-Observation, in [21]): This property implies that an attacker cannot intercept the user's input from inside the user's device (e.g., by a malware). Tapas as well as token-based (and web-based) managers that store the password encrypted might mistakenly appear to be secure in case of the device (or the online password manager server) gets compromised. However, note that once the device gets compromised, the security falls back to the security of the master/nominal password. In that case, commercial password managers that require one single master password to protect all accounts, offer lower level of security (the attacker can compromise "all" the accounts that are served by the password manager). In contrast in SPHINX no information about the pwd and rwd is leaked from the device, since pwd is not entered into the device and rwd is not stored on the device.

S6. Resilient-Upon-Theft (S8 in [21]). This property applies to token/device-based managers and implies that an attacker who can get physical access to the device, temporarily or permanently cannot obtain the passwords. The security arguments under this property are exactly the same as the previous item.

S7. Resistance-to-MITM over Device-Client Channel (captured in S5, Resilient-to-Internal-Observation, in [21]): This metric applies to all token-based and web-based password managers. SPHINX is the only password manager does not require any requirement on device-client channel confidentiality[4] and therefore is secure against MITM attack and eavesdropping. Tapas creates an authenticated channel between the Wallet (device) and the Manager (browser) (via pairing, for example), and then only it can provide the required security against MITM. Traditional web-based password managers protect against MITM attack only through the use of SSL channels (which might be compromised) between the web-based password manager and the client. In traditional token-based password managers the users are required to manually read and enter the password from the device, the manual-human based channel is typically considered secure against the MITM attack, however, physical observation over this channel is possible and is captured in the next item (S9).

S8. Resilient-to-Physical-Observation (S1 in [21]). This property implies that the attacker cannot impersonate a user after observing him/her during the authentication process earlier. These attacks include shoulder surfing, filming the keyboard, recording keystroke sounds, or thermal imaging of keypad. All of the listed commercial password managers require the user to enter the master password and therefore are not resilient to physical observation. Although Passpet, PwdHash and Password Multiplier fill in the password field with the randomized password, they still require the user to enter the master secret. SPHINX, however, is secure to physical observation, since our primary design requires the user to approve a dialog on the device before reconstructing password and filling the password field. To improve the usability of the SPHINX system, the user can optionally disable the need for approval (zero interaction),

4. Authenticated channel between the device and the client is required only if the client and server communicate over TLS/SSL.

however, in that case SPHINX would not be Resilient-to-Physical-Observation.

Usability Metrics

U1. Memorywise-Effortless (U1 in [21]). We call a system Memorywise-Effortless if the user is not required to remember any secrets except for a password or a master password per service. With our definition, all the studied managers are Memorywise-Effortless.

U2. Password-Update-Not-Necessary. SPHINX and all the hash-based managers in our list (PwdHash, Password Multiplier, and Passpet) compute a cryptographic function of the password that the user enters to the web-sites. Hence, they require the user to update their original password when moving over to one of these schemes from a password-only scheme.

U3. Scalable-for-Users (U2 in [21]). The schemes that do not require extra effort from the users' side when the number of accounts are increased are considered as scalable. Scalability is defined from the users point of view and not from the system deployment perspective. With this definition, all the studied password manager are scalable, since users' effort does not increase with increase in the number of accounts.

U4. Physically-Effortless (U4 in [21]). The schemes that do not require user to do anything beyond typing one password, and perhaps clicking a button to activate or run the password manager service are "Physically-Effortless". SPHINX requires the user to tap the approve button on the device, which is a design choice for extra security. SPHINX can work as a service on the phone and would not necessarily require this physical approval from the user-side (at the loss of some security properties). Other hash-based password managers are Physically-Effortless. However, commercial token-based, client-based, and web-based password managers are not effortless since they require the user to transfer the password from the device to the client in addition to typing the master password (unless they offer an additional feature of form auto filling). Since SPHINX does not store either pwd and rwd, such form auto filling by storing the passwords is not recommended in our approach, although it is possible.

Deployability Metrics

D1. Client-Compatible (D4 in [21]). The schemes that do not require any changes at the client-side are called Client-Compatible. This includes schemes that do not require any software or browser extension. Except for the Firefox password manager and web-based commercial password manager, none of the other studied scheme are Client-Compatible.

D2. Nothing-to-Carry (U3 in [21]). The schemes that do not necessarily require additional hardware to operate are in this category. Device-aided schemes would not satisfy this property. However, device-aided schemes increase the security of the system, since an attacker who knows the master password, requires the second factor to authenticate to the service. SPHINX is a device-aided approach. The current implementation of SPHINX uses a smartphone as the secondary device. Tapas is also a device-aided password manager. Token-based password manager also require extra hardware or device.

4.3 Summary of Comparison

SPHINX is different from traditional password managers in many ways. Traditional password managers, such as [1], [12], store the passwords encrypted with a single master password. In particular, with existing device-based managers, the user types in her master password on the application that unlocks the respective password and displays it on the screen. The user then copies the password over to the login-page to authenticate to the service. These managers require a confidential channel (e.g., resistant to shoulder-surfing) between the password manager app and the client, and are also open to dictionary attacks upon application compromise. In addition, an attacker could guess the master password in an online guessing attack against the device. Similarly a potentially malicious code running on the device could learn the user's master password as it is entered into the device and learn all stored passwords. Such password managers normally do not provide resistance against offline dictionary attack in the event of server compromise unless the user picks a random password for each site (i.e., the manager does not enforce a policy). Resistance against phishing is not provided by these managers.

In contrast, SPHINX does not require a confidential channel, and *nothing* (in an information-theoretic sense) is learned about the password by the device (or a malicious software running on the device) or over the device-client channel, even without any encryption of this channel. Finally, by randomizing and hardening the passwords, SPHINX imposes resistance against server-side offline dictionary attack.

SPHINX is also different from "hash-based password managers" that compute a cryptographically hash of a memorable password on the fly. An example of such password manager solutions is PwdHash [39]. PwdHash maps a low entropy password to a randomized one by hashing a (password, domain-name) pair and registering it as the password with the server. PwdHash deterministically transforms a user's password into a more complex password but, unlike our scheme, this transformation does not help against offline dictionary attacks at a compromised server. Moreover, if a user uses the same master password pwd with PwdHash for different services, the compromise of a single server leads to the discovery of pwd via an offline attack and then to the (deterministic) calculation of all rwd's derived from pwd.

5 Discussion

Online SPHINXService. Our current implementation of SPHINX is geared for smartphones. However, it is by no means limited to that. Since SPHINX does not require a confidential channel between D and C and since the device is oblivious to the user's password (pwd or rwd), one possibility is to outsource the device functionality to a remote online (third-party) service, to give rise to an online manager (Online SPHINX). For such an online setting, in contrast to other online password managers, SPHINX would provide optimal resistance in the event of manager compromise since only OPRF key is stored on the online service and is independent of user's password (in particular, this password is not needed for authenticating the user to the online SPHINX).

The online SPHINX service can act as a full replacement of the smartphone as an independent password manager. Online SPHINX can also serve as the password manager to

allow for login from the smartphone itself, which in our current setting (smartphone instantiation) is not supported. Finally, the online service can also be a complement to the smartphone instantiation for backup purposes (see next discussion item).

Online SPHINX can also benefit from a distributed service in which the OPRF keys for web accounts is distributed among several servers. This prevents learning specific keys upon the compromise of one server or a subset of servers (with a threshold scheme). It also provides better "availability" guarantees. OPRF computation in SPHINX can be thersholdized using polynomial secret sharing.

Further work is warranted to carefully design an online version of SPHINX, which can retain all the security and usability advantages of SPHINX while providing increased service availability. Our assumption is that such setting will provide the same security features as SPHINX in Zero Interaction mode, while it may provide better usability since the reliance on the smartphone would not be needed. In a real-world system, both online and smartphone implementations can be offered to the users as a holistic solution, similar to many currently-deployed commercial password management services.

Key Back-Up and Device Upgrade. One natural concern about any smartphone password manager, like the smartphone-based instantiation of SPHINX, is the permanent loss of the phone. To deal with such situations, SPHINX users should back-up the OPRF key on an external storage. When using a new device, this key can then be recovered from the back-up device. Similarly, when upgrading to a new device (e.g., when buying a new phone), the key from the old phone or the back-up device can be copied over to the new device. Such a key transfer should be performed over a secure channel (e.g., a wired connection between the devices). The current device-based password managers also recommend backing up the list of stored passwords on the device in the same way. Even the current two-factor authentication (2FA) systems recommend similar strategies. SPHINXonline service could be used to facilitate such a back-up (as discussed above).

Client Compromise. Password managers are not intended, designed or capable of resisting against client side compromise. Indeed, none of the password managers that we studied are secure against client compromise. The reason is that the password is entered in the webpage manually by the user or automatically by the password manager. In either case, an attacker who resides on a client can theoretically intercept the password. Malicious code and key-loggers are always a threat to browsers in spite of security enhancements in the browsers. However, in our SPHINX system, because we use a "key-ed" password hardening scheme, an attacker who learns pwd using a key-logger can *not* succeed in logging into the web service. An attacker, who compromises the client machine and get it to execute a malicious code, can obtain rwd of an on-going session, and thereby succeed in logging to *only* that service, even if the user uses the same pwd for all services.

One potential solution to offer security under client compromise could be to employ a 2FA mechanism. Since 2FA mechanisms require a password as well as a one-time PIN code produced by the device (the second factor), they can offer better resistance in the event of client compromise (key-logging one PIN code from the client machine will not be sufficient for the attacker to log in over *new* sessions). Since 2FA and SPHINX both use a device during the authentication process, it seems natural to integrate the two schemes together so as to achieve all the security advantages provided by the latter and the resilience to client compromise offered by the former. Such integration may work transparently with current web services that already use 2FA as a means to login users. However, a thorough future investigation is necessary to formalize and realize such SPHINX-enhanced 2FA mechanisms.

Preliminary Usability Evaluation. We conducted an initial small-scale usability study of our SPHINX smartphone manager in lab settings with participants of different educational backgrounds but none skilled in computer security, and other small-scale study in real-life environments with expert users (skilled in computer security), while these users logged into popular daily used web services of their choice (e.g., Gmail, Facebook and Yahoo) using their own accounts. The study followed a methodology in line with that of prior notable studies of password managers [23], [33]. While these initial studies should be expanded in future work, our preliminary results suggest that users' perception of SPHINX security and usability is high, and that SPHINX user experience is satisfactory when compared to password authentication using a human-memorable password (our study's baseline).

6 Conclusions

Passwords are a "necessary evil". In this paper, we attempted to respond to the growing security and usability problems with passwords by proposing SPHINX, a cryptographic password manager that can address most security and usability problems with passwords from the client/user side alone (i.e., transparent to most existing web authentication services). SPHINX is a password management approach, built atop an existing oblivious PRF (OPRF) scheme, that transforms a human-memorable password into a random password with the aid of a device without the need to store the passwords on the device. SPHINX offers several key security guarantees, namely, resistance to: (1) online guessing attacks, (2) offline dictionary attacks under server compromise, (3) offline dictionary attacks under device compromise, (4) phishing attacks, and (5) eavesdropping and man-in-the-middle attacks on the device-client channel. SPHINX also boasts to provide almost the same level of user experience as that of authentication using an easy to memorize password. Unlike other password managers, SPHINX perfectly hides passwords and the master password from itself, and thus remains secure under the realistic threat of the compromise of password managers. Also, unlike other password managers, SPHINX does not require a confidential device-client channel. At the same time and like many other password managers, SPHINX can resist online guessing, offline dictionary under web service compromise and phishing attacks. We designed and implemented a smartphone-based instantiation of SPHINX. Our performance and analytical evaluation of this instantiation shows that it is efficient, highly secure, likely simple to use, and easy to deploy in practice.

ACKNOWLEDGMENTS

This submission is an extension to our previous work [40].

REFERENCES

[1] 1Password: Simple, Convenient Security. [Online]. Available: https://1password.com/, Accessed 2019.
[2] Anonymous hackers claim to leak 28,000 PayPal passwords on global protest day. [Online]. Available: http://goo.gl/oPv2h, Accessed 2019.
[3] Blizzard servers hacked; emails, hashed passwords stolen. [Online]. Available: http://goo.gl/OTNWJC, Accessed 2019.
[4] Current Android Malware. [Online]. Available: http://goo.gl/0sWbXz, Accessed 2019.
[5] Dashlane Password Manager. [Online]. Available: https://www.dashlane.com/, Accessed 2019.
[6] Fine the source of your leaks. [Online]. Available: https://www.leakedsource.com/, Accessed 2019.
[7] Hackers compromised nearly 5M Gmail passwords. [Online]. Available: http://goo.gl/IRu07u, Accessed 2019.
[8] LinkedIn Confirms Account Passwords Hacked. [Online]. Available: http://goo.gl/AWB5KC, Accessed 2019.
[9] Nanohttpd java app. [Online]. Available: https://nanohttpd.com, Accessed 2019.
[10] One Year Of Android Malware. [Online]. Available: http://goo.gl/2UkUJS, Accessed 2019.
[11] Password Manager - Remember, delete, change and import saved passwords in Firefox. [Online]. Available: https://goo.gl/Qve4l7, Accessed 2019.
[12] Password Manager, Auto Form Filler, Random Password Generator & Secure Digital Wallet App. [Online]. Available: https://lastpass.com/, Accessed 2019.
[13] RoboForm: World's Best Password Manager. [Online]. Available: https://www.roboform.com/, Accessed 2019.
[14] RSA breach leaks data for hacking securid tokens. [Online]. Available: http://goo.gl/tcEoS, Accessed 2019.
[15] RSA SecurID software token cloning: A new how-to. [Online]. Available: http://goo.gl/qkSFY, Accessed 2019.
[16] Russian Hackers Amass Over a Billion Internet Passwords. [Online]. Available: http://goo.gl/aXzqj8, Accessed 2019.
[17] A. Adams and M. A. Sasse, "Users are not the enemy," *Commun. ACM*, vol. 42, no. 12, 1999.
[18] M. Bellare, D. Pointcheval, and P. Rogaway, "Authenticated key exchange secure against dictionary attacks," in *Proc. Int. Conf. Theory Appl. Cryptographic Tech.*, 2000, pp. 139–155.
[19] D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange, "Elligator: Elliptic-curve points indistinguishable from uniform random strings," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 967–980.
[20] J. Bonneau, "The science of guessing: Analyzing an anonymized corpus of 70 million passwords," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 538–552.
[21] J. Bonneau, C. Herley, P. C. Van Oorschot, and F. Stajano, "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 553–567.
[22] R. Chatterjee, J. Bonneau, A. Juels, and T. Ristenpart, "Cracking-resistant password vaults using natural language encoders," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 481–498.
[23] S. Chiasson, P. C. van Oorschot, and R. Biddle, "A usability study and critique of two password managers," in *Proc. 15th Conf. USENIX Secur. Symp. - Vol. 15*, 2006, Art. no. 1.
[24] I. Dacosta, M. Ahamad, and P. Traynor, "Trust no one else: Detecting MITM attacks against SSL/TLS without third-parties," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2012, pp. 199–216.
[25] W. Ford and B. S. Kaliski Jr , "Server-assisted generation of a strong secret from a password," in *Proc. 9th IEEE Int. Workshops Enabling Technol.: Infrastructure Collaborative Enterprises*, 2000, pp. 176–180.
[26] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold, "Keyword search and oblivious pseudorandom functions," in *Proc. 2nd Int. Conf. Theory Cryptography*. 2005, pp. 303–324.
[27] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: Validating ssl certificates in non-browser software," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 303–324.
[28] M. Gollam, B. Beuscher, and M. Durmuth, "On the security of cracking-resistant password vaults," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1230–1241.
[29] J. A. Halderman, B. Waters, and E. W. Felten, "A convenient method for securely managing passwords," in *Proc. 14th Int. Conf. World Wide Web*, 2005, pp. 471–479.
[30] S. Jarecki, A. Kiayias, and H. Krawczyk, "Round-optimal password-protected secret sharing and T-PAKE in the password-only model," in *Proc. Int. Conf. Theory Appl. Cryptology Inf. Secur.*, 2014, pp. 233–253.
[31] S. Jarecki, H. Krawczyk, M. Shirvanian, and N. Saxena, "Device-Enhanced Password Protocols with Optimal Online-Offline Protection," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 177–188. [Online]. Available: http://eprint.iacr.org/2015/1099
[32] A. Karole, N. Saxena, and N. Christin, "A comparative usability evaluation of traditional password managers," in *Proc. 13th Int. Conf. Inf. Secur. Cryptology*, 2010, pp. 233–251.
[33] A. Karole, N. Saxena, and N. Christin, "A comparative usability evaluation of traditional password managers," in *Proc. Int. Conf. Inf. Secur. Cryptology.*, 2010, pp. 233–251.
[34] M. R. Karthiga and M. K. Aravindhan, "Enhancing performance of user authentication protocol with resist to password reuse attacks," *Int. J. Comput. Eng. Res.*, vol. 2, no. 8, 2012.
[35] Z. Li, W. He, D. Akhawe, and D. Song, "The emperor's new password manager: Security analysis of web-based password managers," in *Proc. 23rd USENIX Conf. Secur. Symp.*, 2014, pp. 465–479.
[36] M. Mannan and P. C. van Oorschot, "Using a personal device to strengthen password authentication from an untrusted computer," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2007, pp. 88–103.
[37] D. McCarney, D. Barrera, J. Clark, S. Chiasson, and P. C. van Oorschot, "Tapas: Design, implementation, and usability evaluation of a password manager," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, 2012, pp. 89–98.
[38] R. Morris and K. Thompson, "Password security: A case history," *Commun. ACM*, vol. 22, no. 11, pp. 594–597, 1979.
[39] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell, "Stronger password authentication using browser extensions," in *Proc. Proc. 14th Conf. USENIX Secur. Symp. - Vol. 14*, 2005, p. 2.
[40] M. Shirvanian, S. Jarecki, H. Krawczyk, and N. Saxena, "Sphinx: A password store that perfectly hides passwords from itself," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 1094–1104.
[41] F. Stajano, "Pico: No more passwords! in *Proc. Int. Workshop Secur. Protocols*, 2011, pp. 49–81.
[42] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, "Crying wolf: An empirical study of ssl warning effectiveness," in *Proc. 18th Conf. USENIX Secur. Symp.*, 2009, pp. 399–416.
[43] L. Whitney, "LastPass CEO reveals details on security breach," 2011. [Online]. Available: https://goo.gl/WyNjjb
[44] J. Yan, A. Blackwell, R. Anderson, and A. Grant, "Password memorability and security: Empirical results," *IEEE Secur. Privacy*, vol. 2, no. 5, pp. 25–31, Sep./Oct. 2004.
[45] K.-P. Yee and K. Sitaker, "Passpet: convenient password management and phishing protection," in *Proc. 2nd Symp. Usable Privacy Secur.*, 2006, pp. 32–43.
[46] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 95–109.

Maliheh Shirvanian is a Staff Research Scientist in the System Security team at Visa Research. Her main research interests are authentication, system security, and user-centered security. She has several publications in top venue security conferences such as CCS and NDSS. She received her PhD in 2018 from the University of Alabama at Birmingham where she was affiliated with the Security and Privacy In Emerging computing and networking Systems (SPIES) research group. She was the recipient of UAB College of Arts and Sciences 2018 Dean's Award and UAB Departmental Outstanding Student Award. Currently, she is working on improving the security of usable authentication schemes.

Nitesh Saxena is a Professor of Computer Science at the University of Alabama at Birmingham (UAB), and the founding director of the Security and Privacy in Emerging Systems (SPIES) group/lab. He works in the broad areas of computer and network security, and applied cryptography, with a keen interest in wireless and mobile device security, and the emerging field of usable security. Saxena's current research has been externally supported by multiple grants from NSF and NIJ, and by gifts/awards/donations from the industry, including Google (2 Google Faculty Research awards), Cisco, Comcast, Intel, Nokia and Research in Motion. He has published over 120 journal, conference and workshop papers, many at top-tier venues in Computer Science, including: IEEE Transactions, ISOC NDSS, ACM CCS, ACM WWW, ACM WiSec, ACM ACSAC, ACM CHI, ACM Ubicomp, IEEE Percom, IEEE ICME and IEEE S&P. On the educational/service front, Saxena currently serves as the director and principal investigator for the UAB's Scholarship for Service (SFS) program and a co-director for UAB's MS program in Computer Forensics and Security Management. He serves as an Associate Editor for flagship security journals, IEEE Transactions on Information Forensics and Security (TIFS), and Springer's International Journal of Information Security (IJIS). Saxena's work has received extensive media coverage, for example, at NBC, MSN, Fox, Discovery, ABC, Bloomberg, MIT Tech Review, ZDNet, ACM TechNews, Yahoo! Finance, Communications of ACM, Yahoo News, CNBC, Slashdot, Computer World, Science Daily and Motherboard.

Stanislaw Jarecki is a professor of Computer Science at the University of California, Irvine. He received his PhD degree from MIT in 2001. His research focuses on cryptography, applied cryptography, distributed algorithms, fault-tolerance, and privacy.

Hugo Krawczyk is an IBM Fellow and Distinguished Research Staff Member with the Cryptography Group at the IBM T.J. Watson Research Center. He has contributed to the cryptographic design of numerous Internet standards, particularly IPsec, IKE, and SSL/TLS, and is a co-inventor of the HMAC message authentication algorithm. His most recent work in this area includes designs for TLS 1.3, the next generation TLS, and HKDF, the emerging standard for key derivation adopted by TLS 1.3, Signal, WhatsApp, Facebook Messenger and more. Hugo is an Associate Editor of the Journal of Cryptology, a Fellow of the International Association of Cryptologic Research (IACR), and an IBM Fellow. He is the recipient of the 2015 RSA Conference Award for Excellence in the Field of Mathematics, the 2018 Levchin Prize for Contributions to Real-World Cryptography and two IBM corporate awards.