SafeKeeper: Protecting Web Passwords using Trusted Execution Environments

Klaudia Krawiecka
Aalto University, Finland
kkrawiecka@acm.org

Arseny Kurnikov
Aalto University, Finland
arseny.kurnikov@aalto.fi

Andrew Paverd
Aalto University, Finland
andrew.paverd@ieee.org

Mohammad Mannan
Concordia University, Canada
m.mannan@concordia.ca

N. Asokan
Aalto University, Finland
asokan@acm.org

ABSTRACT

Passwords are by far the most widely-used mechanism for authenticating users on the web, out-performing all competing solutions in terms of deployability (e.g. cost and compatibility). However, two critical security concerns are phishing and theft of password databases. These are exacerbated by users' tendency to reuse passwords across different services. Current solutions typically address only one of the two concerns, and do not protect passwords against rogue servers. Furthermore, they do not provide any verifiable evidence of their (server-side) adoption to users, and they face deployability challenges in terms of ease-of-use for end users, and/or costs for service providers.

We present SafeKeeper, a novel and comprehensive solution to ensure secrecy of passwords in web authentication systems. Unlike previous approaches, SafeKeeper protects users' passwords against very strong adversaries, including external phishers as well as corrupted (rogue) servers. It is relatively inexpensive to deploy as it (i) uses widely available hardware-based trusted execution environments like Intel SGX, (ii) requires only minimal changes for integration into popular web platforms like WordPress, and (iii) imposes negligible performance overhead. We discuss several challenges in designing and implementing such a system, and how we overcome them. Via an 86-participant user study, systematic analysis and experiments, we show the usability, security and deployability of SafeKeeper, which is available as open-source.

KEYWORDS

Passwords, Phishing, Intel SGX, Trusted Execution Environment

ACM Reference Format:
Klaudia Krawiecka, Arseny Kurnikov, Andrew Paverd, Mohammad Mannan, and N. Asokan. 2018. SafeKeeper: Protecting Web Passwords using Trusted Execution Environments. In *WWW 2018: The 2018 Web Conference, April 23–27, 2018, Lyon, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3178876.3186101

1 INTRODUCTION

Passwords are by far the most widely used primary authentication mechanism on the web. Although many alternative schemes have been proposed, none has yet challenged the dominance of passwords. In the evaluation framework of authentication mechanisms by Bonneau et al. [8], passwords have the best overall *usability*, since they are easy to understand, efficient to use, and don't require the user to carry additional devices/tokens. They also excel in terms of *deployability*, since they are compatible with virtually all servers and web browsers, incur minimal cost per user, and are accessible, mature, and non-proprietary. However, in terms of *security*, passwords are currently a comparatively poor choice. Two critical security concerns, leading to the compromise of a large number of passwords, are: (i) phishing of passwords from users, and (ii) password database breaches from servers.

Phishing attacks are prevalent in the wild (cf. APWG [2], PhishTank [33]), and increasingly use TLS certificates from browser-trusted CAs (see e.g. [30]). While advanced anti-phishing solutions exist [11, 29], and will improve over time, they alone cannot adequately address the password confidentiality problem, because users may unknowingly or inadvertently disclose passwords to malicious servers. Note that an estimated 43–51% of users reuse passwords across different services [13, 17, 22, 39].

Password database breaches are increasingly frequent: hundreds of millions of passwords have been leaked in recent years (e.g. [14, 34]). Users are completely powerless against such breaches. Password breaches are commonly dealt with by asking users to quickly reset their passwords, which is not very effective [19]. Widespread password reuse makes these breaches problematic beyond the sites where the actual leak occurred [22]. Several recent solutions (e.g. [4, 9, 15, 26]) have been proposed to address password database breaches. Some of them (e.g. [4, 9, 26]) make use of hardware-based trusted execution environments (TEEs) on the server side, but none can protect password confidentiality against *rogue* servers (i.e. compromised servers, or malicious server operators).

Designing effective solutions to protect passwords against rogue servers poses multiple technical challenges in terms of security (How to hide passwords from the authenticating server itself? How to rate-limit password testing by the server?); usability (How to minimize the burden on users? How to support login from diverse user devices?); user-verifiability (How to notify users when the solution is active?); performance (How to realize this at scale?); and deployability (How to allow easy/inexpensive integration with popular website frameworks?).

We present SafeKeeper, a comprehensive system for protecting the *confidentiality* of web passwords. Unlike all previous proposals, SafeKeeper defends against both phishing and password database theft, even in the case of rogue servers. SafeKeeper consists of a *server-side password protection service* that computes a cipher-based message authentication code (CMAC) on passwords before they are stored in the database, as mandated by NIST's digital identity guidelines (SP800-63B) [32]. To protect the CMAC key, this computation is performed within a server-side TEE, isolating it from all other software on the server. SafeKeeper uses a novel rate-limiting mechanism to throttle online guessing attacks by rogue servers.

SafeKeeper's client-side functionality, in the form of a web browser addon, enables end users to detect whether a web server is running the SafeKeeper password protection service within a server-side TEE, and to establish a secure channel from their browsers directly to it. This *assures* users that it is safe to enter their passwords as they will be accessible only to the SafeKeeper password protection service on the server. Unlike other client-side assurance approaches, SafeKeeper does not require users to correctly identify the server (e.g. checking URLs or TLS certificates). As long as users correctly recognize SafeKeeper's client-side signaling from the browser addon, and enter their passwords only to SafeKeeper-enabled web servers, confidentiality of their passwords is guaranteed, *even if users misjudge the identity of the server (phishing), or the server is malicious/compromised (rogue server).* As such, SafeKeeper may present a significant shift in phishing avoidance and password protection.

Our design considers *deployability* as a primary objective. We demonstrate this by developing a fully-functional implementation of SafeKeeper using Intel's recent Software Guard Extensions (SGX), and integrating this with minimal software changes into `PHPass`, the password hashing framework used in popular platforms like WordPress, Joomla, and Drupal, which account for over 34% of the Alexa top 10-million websites [37]. SafeKeeper's client-end functionality does not depend on any additional device/hardware features, and can thus be implemented in most user devices/OSes, including smartphones. Our implementation is available as open-source software [25]. Our contribution is SafeKeeper, including its:

- **Design**: As a password-protection scheme featuring
  - a **server-side password protection service** using off-the-shelf trusted hardware to protect users' passwords, *even against rogue servers* (Sections 4.1–4.3), and
  - a novel **client-side assurance mechanism** allowing users to easily determine if it is safe to enter passwords on a web page (Section 4.4). Our mechanism relies only on verifying whether the server runs SafeKeeper *without having to verify the server's identity or correct behavior.*
- **Implementation and integration:** A full open-source implementation of (i) server-side functionality using Intel SGX, and integration into `PHPass` to support several popular web platforms, and (ii) client-side functionality realized as a Google Chrome browser addon (Section 5).
- **Analysis and evaluation:** A comprehensive analysis of security guarantees (Section 6.1), an experimental evaluation of performance (Section 6.3), deployability in real-world platforms (Section 6.4), and validation of effectiveness of the client-side assurance mechanism via an *86-participant user study* (Section 6.2).

2 PRELIMINARIES

2.1 Storing Passwords

A widely-used approach for storing passwords is for the server to compute a *one-way* function (e.g. cryptographic hash) on the password, and store only the result in the database. When the user logs in, the same function is applied to the supplied password, and the result is compared to the value in the database. An adversary who obtains the database cannot reverse the one-way function, but can guess candidate passwords and apply the same one-way function to test his guesses. Since passwords are weak secrets (e.g. compared to cryptographic keys), a brute force guessing attack is often feasible. The adversary can speed up this attack using *rainbow tables* – pre-computed tables of hashed passwords. If multiple users choose the same password, the results of the one-way function will be the same. To defend against rainbow tables and avoid revealing duplicate passwords, it is customary to use a *salt* – a random number unique to each user that is concatenated with the password before being hashed. However, since salt values are stored in the database, an adversary who obtains this database can still mount brute-force guessing attacks against specific users. Recently (June 2017), NIST updated its digital identity guidelines in Special Publication 800-63B [32]. One of the changes is to mandate the use of *keyed* one-way functions, such as CMAC, for protecting stored passwords.

2.2 Intel Software Guard Extensions

Intel's Software Guard Extensions (SGX) is a recent technology available in desktop and server CPUs [21]. The new SGX CPU instructions allow a userspace application to establish a hardware-enforced TEE, called an *enclave*. The enclave runs in the application's virtual address space, but after it has been initialized, only the code inside the enclave is allowed to access enclave data. The application can call enclave functions (called `ecalls`) via well-defined entry points. Enclave data is stored in a special region of memory, called the Enclave Page Cache (EPC), which can only be accessed by the CPU. When any enclave data leaves the CPU (e.g. is written to DRAM), it is encrypted and integrity-protected, using a key accessible only to the CPU [16]. The enclave's data is therefore protected against privileged software (including the OS/hypervisor), and hardware attacks (e.g. snooping on the memory bus). During enclave initialization, the CPU *measures* the enclave's code and configuration, which constitute the enclave's identity (i.e. its `MRENCLAVE` value). The enclave can *seal* data by encrypting it with a CPU-protected key that can only be accessed from enclaves running on the same CPU with precisely the same `MRENCLAVE` value. Remote attestation is the process through which one party, the *verifier*, can ascertain the precise hardware and software configuration of a remote party, the *prover*. The objective is to provide the verifier with sufficient information to make a trust decision about the prover. SGX supports remote attestation by providing verifiers with a signed *quote* from the enclave, which includes the enclave's precise identity (`MRENCLAVE` value) and the enclave's public key [1]. The verifier can validate this quote using the *Intel Attestation Service* (IAS), and can then establish an end-to-end encrypted channel directly to the enclave. SGX remote attestation is explained in detail in our technical report [24].

3 SYSTEM MODEL AND REQUIREMENTS

3.1 Overview

We use the term *password* to refer to any user-memorized authentication secret. Passwords are generally weak secrets [7], which are often re-used across multiple services. Figure 1 is a generalized model of a password-based authentication system. With the assistance of client-side software (e.g. a web browser), a user sends her user ID and password to a server over a secure channel (TLS), and based on this, the server makes an authentication decision. Although real systems are undoubtedly more complicated, they can all be logically represented as the model in Figure 1. Therefore, we use this model and terminology throughout this paper.

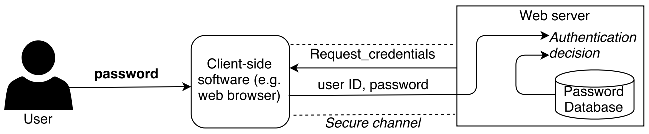

Figure 1: Generalized model of a password-based authentication system.

Confidentiality of user passwords may be compromised in multiple ways, including:

- Users disclose a password directly to the adversary under the mistaken impression that the password is being sent to the intended service (e.g. phishing attacks).
- Information about the password, or the password itself, is leaked from the server (e.g. stolen password database).
- Server compromise (e.g. web server memory snooping).

Any comprehensive solution for protecting passwords must defend against *all* these different attack avenues.

3.2 Adversary Model

The goal of our adversary ($\mathcal{A}dv$) is to learn users' plaintext passwords. Our adversary model is stronger than that of previous solutions – we allow for the possibility that $\mathcal{A}dv$ may have the capabilities of the *operator* (owner/administrator) of the web server itself. This covers both compromised web servers, as well as malicious server operators (for clarity, we use the term *rogue server* to refer to both of these). The widespread practice of reusing the same or similar passwords across services provides the incentive for $\mathcal{A}dv$ to learn plaintext passwords, which they can then abuse in different ways, including (i) masquerading as the user on a *different* service, where the user may have used the same or similar password; (ii) accessing user data encrypted by a password-derived key (e.g. for encrypted cloud storage services, especially, if forced legally or illegally); and (iii) selling/leaking user passwords to other malicious entities. Concretely, we allow $\mathcal{A}dv$ the following capabilities, covering rogue servers and weaker external adversaries:

Access password database: $\mathcal{A}dv$ has unrestricted access to the password database. This models both rogue servers and weaker external adversaries who may steal the password database.

Modify web content: $\mathcal{A}dv$ can arbitrarily modify the content sent to the user (including active content such as client-side scripts). This also models weaker external adversaries who can modify content using attacks such as cross-site scripting (XSS).

Access to server-client communication: $\mathcal{A}dv$ can read all content sent to the web server, including content encrypted by a TLS session key. For a rogue server, such access is easily obtained.

Execute server-side code: $\mathcal{A}dv$ has full knowledge of all software running on the server, and is able to execute arbitrary software on the server. This captures a powerful attacker who gains access and escalates privileges on the server, or a malicious server operator.

Launch phishing attacks: $\mathcal{A}dv$ can launch state-of-the-art phishing attacks, including targeted attacks.

We assume that $\mathcal{A}dv$ *cannot* compromise the client-side software, including the user's OS and browser, but can send any content to the client. Although client-side security is important, it is an orthogonal problem and is addressed by other means (e.g. client-side platform security). We focus on how password-based authentication can be made resilient against strong server-side adversaries.

We assume $\mathcal{A}dv$ is computationally bounded, and thus cannot feasibly subvert correctly-implemented cryptographic primitives. We also assume $\mathcal{A}dv$ does not have sufficient resources to subvert the security guarantees of hardware-based TEEs through direct physical attacks. Denial of service (DoS) attacks are out of scope because a rogue server can always mount a DoS attack by simply refusing to respond to requests.

3.3 Requirements and Objectives

Given our strong adversary model, a full server compromise could undoubtedly cause significant damage (e.g. theft, loss, or modification of user information). Our aim is to guarantee that even such a compromise will not leak users' passwords. This is a valuable security guarantee because passwords are often re-used across multiple services, or used to derive cloud storage encryption keys.

However, guaranteeing the confidentiality of stored passwords is not sufficient to defend against rogue servers or phishing attacks. Thus we also need to provide users with the means to easily and effectively determine when it is safe to enter their passwords. Therefore, we define the following two requirements for a comprehensive solution for protecting a password-based authentication system:

R1 **Password protection:** The server must protect users' passwords by fulfilling all the following criteria:
  (i) The strong adversary defined above cannot obtain users' passwords through any means other than guessing (e.g. he cannot observe passwords in transit or while they are being processed on the server).
  (ii) *Offline* password guessing must be computationally infeasible, irrespective of the strength of the password.
  (iii) *Online* password guessing must be throttled, irrespective of the adversary's computational capabilities.

R2 **User awareness:** End users must be able to easily and accurately determine whether it is safe to enter their passwords when prompted by a given server (i.e. indirectly determine whether the server fulfils Requirement R1).

Note that Requirement R2 *does not* mandate users to understand the precise technical security guarantees of Requirement R1, but rather that the solution should enable users to determine *which* servers meet this requirement, and will thus protect passwords.

In order to be effective, any solution for protecting passwords must be deployable in real-world systems. Therefore, in addition to the above two security requirements, we also define the following deployability goals:

Minimal performance overhead: The solution should not noticeably degrade the performance of password-based authentication systems, either in terms of *latency* (the time required to complete a single authentication attempt), or *scalability* (the overall rate at which authentication attempts can be evaluated).

Minimal software changes: It should be possible to integrate the solution into a wide range of existing software systems without requiring significant effort.

Ease of upgrade: It should be possible to transparently upgrade existing password-based authentication systems (e.g. without requiring users to reset their passwords). Existing mechanisms for changing/resetting passwords should also remain unaffected.

4 DESIGN

As shown in Figure 2, SafeKeeper consists of a *server-side password protection service*, which computes a cipher-based message authentication code (CMAC) on passwords before they are stored in the database. An adversary must obtain the CMAC key in order to perform offline guessing attacks against a stolen password database. In SafeKeeper, this key is randomly generated and protected within a server-side Trusted Application (TA), executing within the TEE; see Section 4.1.

With direct access to the password protection service, a rogue server (i.e. a compromised server or malicious server operator) can also perform online password guessing attacks. In this case, the adversary supplies a guessed password, and the password protection service returns the processed result, which the adversary compares against the stored value. To defend against this attack, the password protection service must limit the rate at which it processes passwords. SafeKeeper achieves this by enforcing rate limiting in the TEE; see Section 4.2.

Furthermore, the rogue server may attempt to observe passwords *before* they are sent to the password protection service. A secure channel between the web browser addon and the server (e.g. a TLS connection) is insufficient as the server-end of such a connection is controlled by the server operator. Instead, SafeKeeper establishes an end-to-end secure channel between the browser and the TEE-protected password protection service; see Section 4.3.

Finally, the browser needs some way to determine which input data should be sent via this secure channel to the password protection service (e.g. passwords but not user IDs). To improve usability, the server operator defines which input fields will be protected, and then the SafeKeeper browser addon displays this information to the user. The user is thus only required to validate that the password field is protected; see Section 4.4.

4.1 Server-side Password Protection

The SafeKeeper password protection service (SafeKeeper TA) is designed to be a *drop-in replacement* for existing password hash algorithms. As such, it takes as input the concatenation of the password and the corresponding salt value and outputs a CMAC, which the server stores in its database. To protect the CMAC key, SafeKeeper computes the CMAC inside a server-side TEE. The TEE provides strong isolation (e.g. hardware-enforced) from all other software on the server (including the OS/hypervisor), and thus ensures that the CMAC key is available only to the TA code. Even if an adversary obtains the password database, he cannot perform an *offline* password guessing attack because he would need the CMAC key to test his guesses against the database. Since offline attacks are no longer possible, SafeKeeper can use any cryptographically secure one-way function, i.e. specifically-designed password hash functions (e.g. [5, 6]) are not essential (but can be used). The adversary is thus forced to try *online* guessing attacks, which are mitigated by SafeKeeper's rate-limiting mechanism.

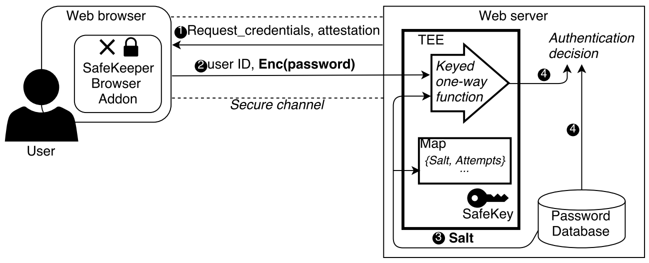

Figure 2: Overview of the SafeKeeper design.

4.2 Rate Limiting

Ideally, the password protection service should perform rate limiting on a per-account basis to: (i) protect each account password, and (ii) avoid rate-limiting a user due to the actions of other users. Note that, in order to serve as a drop-in replacement in existing authentication frameworks, the SafeKeeper TA only computes the CMAC, but does not make or learn the authentication decision. In other words, it cannot distinguish between the two scenarios: (i) when the user account is under guessing attack, and (ii) when a legitimate user is attempting to login multiple times within a short period of time (with or without a valid password). Also, as the CMAC does not take the user ID as input, we cannot implement user ID based rate limiting. Changing the function to include user IDs as input would require non-trivial changes to the server software, limiting SafeKeeper's use as a drop-in replacement. As a work-around, we rate limit each account using the unique per-account salt values, which are provided to the CMAC as part of regular operation. It is general security best-practice to use unique salt values for each account, but if a server operator chooses the same salt for multiple (or all) accounts, he only restricts his own guessing capability. Furthermore, since the salt is a fixed pre-determined length, the adversary cannot perform a type-substitution attack by providing a salt of a different length, e.g. concatenating the first few characters of a guessed password with the salt value.

SafeKeeper limits password processing for each account (salt) based on a *quantized maximum rate*. Simply enforcing a *maximum rate* (e.g. waiting W minutes between password attempts) would negatively impact usability in cases where the user mistyped a password. Instead, the quantized maximum rate allows a fixed number of attempts within a pre-defined time interval, but doesn't mandate a delay between these attempts. For example, each user could be allowed N attempts that could be used at any time within each 24-hour period. After exhausting these attempts, the user has to wait until the following time period, when the count is reset. By calibrating $N = \frac{24 \times 60}{W}$, this achieves the same overall rate as waiting W minutes between attempts, but significantly improves usability when multiple attempts are required in quick succession.

4.3 Remote Attestation

To securely transmit a user password, the SafeKeeper browser addon must correctly authenticate the SafeKeeper password protection service (SafeKeeper TA) running inside a TEE, via remote attestation. In addition, the addon must establish a secure channel directly with the TA. SafeKeeper uses remote attestation to assure the addon of the precise TA running inside the server-side TEE. To verify that it is communicating with a genuine SafeKeeper TA, the addon verifies the quote, and then checks the binary measurement against a whitelist of known SafeKeeper TAs. Since the same TA can be used by many websites, and the functionality of the TA is unlikely to change, the whitelist of genuine SafeKeeper TAs will be short, and can be built into the SafeKeeper addon. The attestation protocol includes a key agreement step through which the browser and the TA establish a shared session key. Note that the attestation protocol provides only unilateral authentication of the SafeKeeper TA towards the client; i.e. the client software or the user is not authenticated during attestation. Thus, anyone, including an adversary can establish a connection and interact with the TA. However, since the key agreement step is cryptographically bound to the TA's remote attestation, the adversary cannot perform a man-in-the-middle attack when legitimate clients communicate with the TA. Using the shared session key, the addon encrypts the password before the page is submitted. The encrypted password is sent to the server in place of the original password, and may be wrapped in additional layers of encryption (e.g. TLS). On the server, the encrypted password is input to the TA, which decrypts it using the shared key and performs the CMAC.

4.4 Client-side Assurance Mechanism

SafeKeeper's client-side assurance mechanism can be added to existing web browsers e.g. by installing a browser addon. This addon executes the remote attestation protocol and establishes a secure channel with the TA. If the attestation succeeds, the addon changes its appearance (e.g. its icon) to signal this to the user.

The server specifies in the web page which input fields should be encrypted and sent to the SafeKeeper password protection TA. The addon parses this information and encrypts any text entered into these fields. However, a rogue server may specify that some non-password fields should be protected, while the actual password field is left unprotected. To prevent this, the SafeKeeper addon signals to the user which input fields are protected by greying out the whole page, highlighting only the text input fields it will encrypt, and displaying an information tooltip.

The adversary could attempt to spoof the highlighting performed by SafeKeeper (e.g. by highlighting fields that are not actually protected). To mitigate this, we use a similar principle to a *secure attention sequence* by requiring the user to click on the browser addon icon to activate the highlighting. This click cannot be detected or prevented by the adversary (as it is outside of the browser DOM). After the user has clicked, the SafeKeeper icon is again changed to indicate that it is in the *highlighting mode*. This provides a spoofing-resistant mechanism for signaling to the user which input fields will be protected. The user is thus assured that a password entered into such an input field will always be protected by SafeKeeper, regardless of the identity of the website or the behavior of the server.

Unlike many other client-side approaches (e.g. password managers), the SafeKeeper browser addon is stateless and user-agnostic. This makes our client-side assurance mechanism a good candidate to be integrated directly into web browsers. We discuss other possible mechanisms in our accompanying technical report [24].

5 IMPLEMENTATION

We have implemented a fully-functional open-source prototype of SafeKeeper [25]. In this section, we describe the specific implementation challenges and our solutions.

5.1 Server-side Password Protection

Our implementation of the password protection service uses Intel's recent Software Guard Extensions (SGX). However, SafeKeeper can use any equivalent TEE that provides isolated execution, sealed storage, and remote attestation. We use SGX for its performance and increasing prevalence on server platforms (e.g. Intel Xeon [20]).

The design of our SGX enclave is kept minimalistic, consisting of only four `ecalls`. When the enclave is started for the first time, the `init()` function uses Intel's hardware random number generator (RDRAND instruction) to generate a new strong CMAC key. When the enclave is later restarted, this function is used to pass previously-sealed data to the enclave. The `process()` function calculates the CMAC on a password and returns the result. We use the Rijndael-128 CMAC function, as this meets our security requirements and can be computed using AES-NI hardware extensions.

We integrated SafeKeeper's password protection service into the `PHPass` library [31], which is widely used for password hashing in popular web platforms including WordPress, Joomla, and Drupal. By default, `PHPass` uses a software implementation of MD5 with 256 iterations. We replaced this with a single call to our enclave, using the PHP-CPP framework [28].

5.1.1 Rate limiting. In addition to the web server's rate limiting (e.g. Captchas after failed attempts), we implement a rate-limiting mechanism within the TEE-protected password service (Section 4.2). The SafeKeeper TA keeps an in-memory map associating each salt (i) with a number of remaining attempts ($attempts_i$). To maximize flexibility, our implementation uses a 64-bit salt and a 32-bit integer for $attempts_i$, although this can be decreased to reduce memory consumption if needed. When `process()` is called for salt i, this function first checks the value of $attempts_i$; if the value is zero, it returns only an error; otherwise $attempts_i$ is decremented by one and the CMAC result is returned. The enclave stores t_{reset}, the time at which all $attempts$ values are reset to a predefined value, $attempts_{max}$. The `reset_attempts()` function first obtains the current time; then, if t_{reset} has passed, sets all $attempts$ values to $attempts_{max}$ and increases t_{reset} by a predefined value. Although the effective rate is set by the enclave developer, it is verified by the user via the browser addon, so a malicious developer cannot set an arbitrarily high rate.

To allow the enclave to be restarted (e.g. if the server is rebooted), the `shutdown()` function securely stores the state information outside the enclave. Specifically, the enclave seals the CMAC key, the map of salts and *attempts* values, and t_{reset}. This sealed data can be restored to the enclave via the `init()` function. The enclave uses hardware-backed monotonic counters to prevent rollback attacks in which the adversary attempts to restore old sealed data.

A rogue server may attempt to reset the *attempts* values by abruptly killing the enclave without first sealing its state. However, the enclave will detect this because the counter value in the sealed data will not match the current value of the hardware monotonic counter. In this case, the enclave has no way of restoring the previous *attempts* values – the data has been irreversibly lost. Therefore, the only secure course of action is to set t_{reset} to some predetermined time in the future, and set all *attempts* to zero (i.e. to impose the maximum penalty). This captures the worst-case scenario in which the adversary had exhausted all guessing attempts against all accounts. Note that, during normal operations, enclave crashes should be rare, and with proper load balancing the effects of abnormal enclave crashes can be amortized.

5.1.2 Remote attestation. Remote attestation is used to assure the browser that it is communicating with a genuine SafeKeeper password protection service running inside an SGX enclave. It is achieved by obtaining a *quote* from the enclave and verifying it using the Intel Attestation Service (IAS). The quote includes an unforgeable representation of the code executing inside the enclave. As described in Section 2, the remote attestation protocol provided with the SGX SDK involves four messages and two round-trips in order to achieve mutual authentication between the verifier and the enclave. In SafeKeeper there is no requirement for the browser addon to authenticate itself to the enclave. Any entity can request a quote from the enclave and then decide whether to establish a secure channel. We designed and implemented an optimized attestation protocol for SafeKeeper. Due to space limitations, we refer readers our technical report [24] for full details of this protocol.

5.2 Client-side Assurance Mechanism

We implement SafeKeeper's client-side assurance mechanism as an addon for the Google Chrome browser (similar implementations can also be developed for the other browsers). We assume that users can download and install this addon securely (e.g. using the Chrome Web Store), and receive software updates when available. Note that browser vendors are actively working to ensure the security of browser addons [27].

Our browser addon is written in JavaScript and consists of two parts: (i) a *background script* implementing the main functionality; and (ii) a *content script* injected into each web page in order to interact with the page content. The addon therefore requires permission to access tabs data, use the browser storage, and capture and modify certain web requests.

5.2.1 Detecting the password protection service. Web servers using SafeKeeper send an SGX quote in the HTTP response header of web pages with protected fields (e.g. the login form). This quote is processed by the addon's background script, which verifies the integrity of the service and the validity of the quote. Specifically, the addon extracts from the quote the unique hash representing the enclave's loaded code (i.e. the `MRENCLAVE` value), and checks if this is included in its list of trusted values. This list can be updated in a similar manner to updating the browser. If the enclave's identity is trusted, the addon validates the quote using the Intel Attestation Service (IAS); see Section 5.1.2. If the attestation process is successful, the background script generates a new DH key pair for the website, and establishes the shared key using the enclave's public key. Finally, the background script changes the addon's icon to indicate the website supports SafeKeeper; see Figure 3.

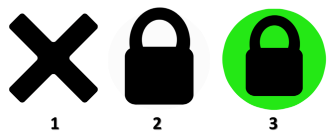

Figure 3: SafeKeeper browser addon icons: (1) SafeKeeper is unavailable on the website, or the attestation protocol has failed; (2) SafeKeeper is supported and a secure channel has been established; (3) SafeKeeper is highlighting the protected input fields (see Section 5.2.2).

5.2.2 Highlighting protected input fields. When a page is loaded, the injected content script checks for the SafeKeeper metatag. If present, this tag specifies which input fields must be encrypted by SafeKeeper. When the user clicks the browser addon icon, a popup window appears to provide information about the current web page. Clicking the addon icon also serves as the secure attention sequence described in Section 4.4, as this cannot be manipulated by the adversary-controlled web page script. This action sends a message to the content script, which modifies the website's DOM elements to highlight the protected input fields and attach an information tooltip to inform the user why each field is highlighted. When the user clicks on the icon again, the content script restores the page's original appearance. When the web page is submitted, the content script encrypts all values from the protected input fields using the shared key agreed with the SafeKeeper password service.

5.2.3 Defending against malicious client-side scripts. As defined in our adversary model (Section 3), a malicious server operator has the capability to modify the content of the web page, including adding client-side scripts. This poses several threats. First, a malicious script may attempt to read the password as it is typed by the user. Client-side scripts that exhibit this behaviour for other types of personal information have already been observed in the wild [18]. With the exception of disabling the malicious script, there is no way to avoid this using current browser addon technologies since the script is executing in the same domain as the text input field. Second, malicious client-side scripts can attempt to spoof the highlighting of input fields performed by the SafeKeeper browser addon. Third, although the adversary cannot detect when the SafeKeeper highlighting has been activated, he can add a time delay to his malicious script in order to spoof the SafeKeeper UI *after* the user has clicked the SafeKeeper icon.

Currently Google Chrome does not provide a direct option to disable client-side scripts for particular websites, so for SafeKeeper we have developed an alternative approach to achieve this using a *Content Security Policy* (CSP). Using the toggle switch on the SafeKeeper popup window, users can disable scripts for individual websites. SafeKeeper then reloads the page and injects our custom CSP metatag into the header, which still allows our injected content script to run, but blocks all other scripts on the page. A limitation of this approach is that there is a race condition between the time our CSP is injected, and the loading of other scripts. However, our experimental evaluation showed that only pages loaded directly from `localhost` were fast enough to evade our CSP. When scripts are disabled for a particular website, the addon stores this information in Chrome's local storage and continues to disable scripts on future visits to this website, until re-enabled by the user.

However, disabling client-side scripts often negatively affects the usability of the web page. Therefore, by default SafeKeeper allows client-side scripts. Careful users can still turn-off all client-side scripts manually when needed. In the accompanying technical report [24], we present alternative approaches that ensure the same level of security without disabling client-side scripts.

6 EVALUATION

6.1 Security Analysis

As defined in Requirement R1, a comprehensive solution for protecting passwords must (i) prevent even the strongest adversary (i.e. rogue server) from observing passwords in transit or during processing on the server; (ii) prevent offline password guessing attacks; and (iii) throttle online password guessing, irrespective of $\mathcal{A}dv$'s computational capabilities. In this section we analyse SafeKeeper's security guarantees against each of these classes of attacks. In our accompanying technical report [24], we further discuss SafeKeeper's resilience against other types of attacks, including roll-back attacks, run-time attacks, and SGX side-channel attacks.

6.1.1 Passwords in transit and on the server. The secure channel, based on the DH key agreement between the browser addon and the enclave, ensures the confidentiality of passwords while in transit over the network, and while they are being processed on the server, before they are input to the enclave. Since this channel is cryptographically bound to the enclave's remote attestation quote, the client-side software is assured that it is communicating with the correct enclave. Note that, this channel's security is independent of any other layer, such as TLS, established between the browser and web server. As such, password confidentiality remains unaffected even if there are flaws in TLS/HTTPS protocols (e.g. [3, 36]).

6.1.2 Offline guessing. In an offline attack, the adversary ($\mathcal{A}dv$) would attempt to guess the password and test his guess directly against the leaked password database, bypassing any online guessing prevention by the web server. $\mathcal{A}dv$ is therefore not subject to rate-limiting, and can perform guesses at the maximum rate supported by his available hardware. However, to test his guesses, $\mathcal{A}dv$ must also guess the CMAC key, and thus making offline attacks infeasible even against very weak passwords.

6.1.3 Online guessing. To avoid having to guess the CMAC key, a malicious server operator could perform an *online* attack by submitting password guesses to the SafeKeeper password protection service, running in the SGX enclave. However, $\mathcal{A}dv$'s rate of guessing is then constrained by SafeKeeper's rate-limiting mechanism described in Section 5.1.1. This increases the difficulty of guessing the password by increasing the time required. For example, if an average strength password provides approximately 20 bits of entropy [7] and SafeKeeper's effective maximum rate were set to $N = 144$ authentication attempts per day, the average time required to guess this password (i.e. 2^{19} guesses) would be nearly 10 years. Note that this still cannot protect very weak passwords (e.g. short passwords or those appearing in lists of frequently used passwords). This is a fundamental limitation of password-based authentication.

Even though our rate limiting is based on the salt, which is under $\mathcal{A}dv$'s control, $\mathcal{A}dv$ cannot increase his guessing rate by manipulating the choice of salts. For example, if $\mathcal{A}dv$ chooses the same salt for all accounts, he can test his guesses against all passwords in the database (e.g. N guesses per day, tested against all accounts). However, if he chooses unique salts he can test the same number of guesses against each individual account, and possibly vary guesses between accounts (e.g. N guesses per day against each account). Additionally, he may try to add fake accounts with guessed passwords to leverage the fact that the same passwords will result in the same CMAC values, if a global salt is used. However, as passwords of new accounts must also be processed by the SafeKeeper password protection service, the number of guessing attempts per day remains unaffected. An external adversary (i.e. without compromising the web server) will also be restricted by any online guessing prevention mechanisms (e.g. Captcha) implemented by an honest server operator.

6.2 Usability Evaluation

To evaluate SafeKeeper against Requirement R2, we conducted a user study for the client-side browser addon. The objectives of this study were to: (i) quantify the ability of participants to use the addon correctly; (ii) assess the *memorability* of the addon usage after a period of disuse; and (iii) analyse the difficulty of using the addon. This user study was carried out in accordance with the standard practices of our institution. No data protection issues arose because participants were not asked to use/disclose any personal data.

6.2.1 Participants and methodology. We recruited 86 participants using institutional mailing lists and social media. The participants were randomly split into two groups: main study group (64 participants), and control group (22 participants). We collected basic demographic information for the main study group: 70% of participants were male and 30% were female; ages between 18 to 39 years; participants' educational qualification: 2% Ph.D., 34% Master's, 41% Bachelor's, 9% High school diploma (14% unspecified).

Main Group: Each participant in this group was initially shown the SafeKeeper information page, which contains the same information a normal user would see when installing our browser addon. Participants were then given a set of 25 websites. These websites were clones of popular websites, created using *HTTrack* [10] and hosted on an internal institutional server. We slightly modified these sites for our experiment; see Table 1. We did not warn the participants about spoofing. The websites were listed in the same order for all participants, but participants could access these in any sequence and had the option to return to previous websites.

Table 1: Test websites

SafeKeeper lock icon	Password protected	Spoofing types	# websites
✓	✓	None	4
✓	✓	Other fields highlighted	5
✗	✗	None	6
✗	✗	Password field highlighted	3
✓	✗	Password field highlighted	4
✗	✗	Password field highlighted after delay	3

For each website, participants were asked: *"Does this website protect your password using SafeKeeper?"* Participants were instructed not to enter any password or other information on the website, but simply to record their answer on the provided paper form. The available options were: *Yes*, *No*, and the level of certainty of the answer. To assess the statistical significance of our results, we assumed a null hypothesis in which participants guess either *Yes* or *No* in a uniformly random manner. Under this null hypothesis, the effectiveness would therefore be 50%. We assumed the standard 5% threshold value for statistical significance ($\alpha = 0.05$).

Follow-up Study: After two months, we invited 20 participants from the main group to participate in a follow-up study, with the objective of measuring how well they remembered how to use SafeKeeper after a relatively long period of disuse. The procedure was the same as for the initial study, except that participants were not shown the SafeKeeper information page, and were not reminded of the instructions for using SafeKeeper. They were shown the same set of websites in a different order and asked the same question.

Control Group: To obtain a baseline against which to assess the results of the follow-up study, we invited approximately the same number of participants (22) to use the tool without any instructions. This was the same procedure as the follow-up study, except that this control group had never previously used the addon. This control group is therefore the best approximation of the scenario in which users have completely forgotten how to use SafeKeeper.

6.2.2 Results. In the main group, participants correctly identified 86.81% of websites. Calculating the p-value for each participant (across all websites) showed that for 80% of participants, $p < 0.001$. The p-value for each website (across all participants) does not exceed 0.001 for 88% of websites.

In the follow-up study, the 20 recalled participants correctly identified 91% of websites on average (the average for these participants was 93% in the initial study). Calculating the p-value for each participant using McNemar's test shows that for 95% of participants, the p-value exceeds 0.2. This means that there was no statistically significant decrease in the effectiveness of SafeKeeper after two months of disuse. The p-value of only one participant shows a statistically significant decrease in effectiveness.

Even with no instructions, the control group participants correctly identified 74% of websites on average. Using Fisher's exact test to compare the follow-up group to the control group resulted in $p = 6.4 \times 10^{-14}$ (far smaller than the threshold of 0.05), showing that the difference is statistically significant.

6.2.3 Discussion. Achieving almost 87% effectiveness, we have exceeded the percentage indicated in the null hypothesis, providing evidence of the addon's utility. Out of 64 participants in the main group, only for 5 was $p > 0.05$. Among the 25 websites, only for one website was $p > 0.05$ (the first phishing website participants encountered). The follow-up study indicates that SafeKeeper's effectiveness does not diminish, even after long periods of disuse (only one out of 20 participants showed a statistically significant drop in effectiveness). Surprisingly, the control group managed to use the browser addon without any instructions to correctly identify a relatively high number of websites. We suspect that these participants may have inferred the instructions to a certain extent based on the addon's behaviour and the text displayed in the popup window. Based on the background questionnaires, nearly 58% of participants do not usually check for a secure connection while browsing the web. 75% of participants were aware of phishing. When asked to assess the level of difficulty of using SafeKeeper, participants answers were: "very easy to use" (39%), "easy to use" (55%), "difficult to use" (6%), "very difficulty to use" (0%). Overall, 76% of participants said they would like to use SafeKeeper in their own browsers.

6.3 Performance Evaluation

Using the implementation described in Section 5, we evaluated the *memory consumption* and *scalability* of our server-side password protection service, as well as the *latency* of verifying a quote. All reported performance measurements are the average of 10 trials, using a simulated password database of 1 million active unique users (i.e. 1 million unique salt values), performed on an Intel Core i5-6500 3.20 GHz CPU with 8GB of RAM.

Memory consumption: Our enclave required at most 110 MB of heap memory to store an in-memory rate-limiting map containing all 1 million salt values. As discussed in Section 5.1.1, memory consumption could be reduced by using more compact representations of the salt or counter values. Note that the enclave does not have to store the salts of every user in the database at the same time – it only stores salts it has processed in the current time window (e.g. users who authenticated in the past 24 hours). The memory consumption can thus be further decreased by reducing this time window (e.g. setting the rate to 36 attempts per 6 hours would only require storing salts for the past 6 hours).

Scalability: To measure the performance of the server when multiple users attempt to authenticate concurrently, we instrumented `PHPass` to measure the maximum password processing rate. With the default hash function (software MD5, 256 iterations), the maximum rate is 446 (±10) passwords/second. With the SafeKeeper password protection service, the rate *increases* to 1653 (±70) passwords/second, since we do not require multiple iterations. We also measured the raw performance of the enclave without PHP (e.g. for websites running optimized software): the maximum rate is 101,337 (±4186) passwords/second. Therefore, even for high-volume websites, this is unlikely to be a performance bottleneck.

Latency: The average latency of verifying a quote is 866 ms (±25 ms). However, since this is done asynchronously while the web page loads, and since the time is comparable with the typical loading time of the web page, this should not have any noticeable impact on the user's browsing experience.

6.4 Deployability Evaluation

Minimal software changes: SafeKeeper's server-side password protection service is designed to be a drop-in replacement for the one-way functions used in current password hashing frameworks (e.g. the PHPass library). Integrating SafeKeeper into PHPass required adding one line of code to initialize the enclave, and changing three lines of code in the password processing function.

Existing servers can be migrated to SafeKeeper. Password hashes in an existing database can be input to a SafeKeeper enclave, which performs the CMAC as usual. This results in a so-called *onion hash*. When users log in, their passwords are sent directly to a special SafeKeeper enclave, which first performs the original hash, and then the CMAC. This allows the security of existing password databases to be upgraded without requiring users to provide their passwords.

Due to space constraints, we discuss other deployability aspects in our technical report [24], including backup of the CMAC key, and scaling SafeKeeper to multiple servers for load-balancing.

7 RELATED WORK

Password research has a rich history. Here we focus on proposals that improve password security against password database compromises, emphasizing solutions that leverage hardware-based security enhancements. However, no previous work has considered password confidentiality against rogue servers.

To mitigate offline guessing attacks against hashed passwords, e.g. due to password database breaches, several proposals take advantage of SGX enclaves [4, 9, 26]. These proposals either encrypt or calculate an HMAC of the password inside an enclave. However, they all assume a significantly weaker adversary model in which the server operator is trusted. Specifically, they do not consider the confidentiality of passwords on the server before they are input to the enclave, nor the possibility for a malicious server operator to perform a brute-force guessing attack using the enclave, nor the risk of phishing. SafeKeeper goes beyond these solutions by resisting rogue servers and phishing attacks.

Various non-SGX server-side approaches have been proposed. Facebook uses a *remote password service* [15] that computes a keyed one-way function on the password and returns the result. The secret key used in this function never leaves their service. Even if an adversary obtains the (keyed) password repository, he must guess the secret key, or connect to the Facebook remote password service (easy to detect and rate-limit). However, running such a service and protecting the key sever against attacks may be infeasible, especially with a limited budget. Also, the key service must be trusted (e.g. not to collude with an attacker), but the trustworthiness of the key server cannot be validated by users.

Another server-side approach is to use specialized hardware to establish an isolated execution environment. Cvrcek et al. [12] use a custom-built USB device to store a secret key and use this to calculate HMACs of passwords. Without the USB device, calculating the HMACs is not possible, and thus cracking passwords from leaked tags is also infeasible. Their prototype was capable of computing only 330 HMAC tags per minute, although scalability could be improved using multiple USB devices. However, plaintext passwords remain available to the server operator, and the trustworthiness of the USB device cannot be validated by users.

As an alternative software-only server-side approach, PolyPasswordHasher [38] uses a special set of *protector* account passwords to prevent offline dictionary attacks against the rest of the *shielded* account passwords. For protected accounts, it applies an XOR function to a *share* and the hash of a salted password. The share values are derived using a threshold cryptosystem, and stored only in memory (in plaintext). The result of the XOR operation is stored in a database on the web server. To guess shielded passwords, the adversary must crack a threshold of protector passwords (e.g. 3–5), and collect the corresponding shares. The shielded passwords are encrypted using a secret, not the shares, and then stored in the database. Therefore, the shielded accounts that use weaker passwords will not leak the information about the shares. The security of this solution relies on the passwords chosen by the protectors (e.g. admin accounts).

Users often remain logged into multiple services at the same time. SAuth [23] leverages this to detect password database compromise. A login attempt to a target site must be validated by a separate *vouching* site (e.g. to log into Gmail, the user must be logged into Facebook). Thus, an attacker who compromises passwords from one site will be unable to use them unless he can also compromise passwords from the vouching site (assumed to be unlikely).

Several client-side techniques have also been proposed. For example, the PwdHash [35] browser addon applies a pseudorandom function to the concatenation of a password and a salt, on the client side. The salt is generated based on a website domain, which binds the password to the specific website. However, an adversary who compromises the web server knows the domain name, and can therefore perform an offline brute-force guessing attack.

Compared to existing work, SafeKeeper provides additional security features, including: protecting password confidentiality against a malicious server operator, strict password guess rate-limiting from within our enclave, enabling users to validate SGX protections of their passwords via browser UI, and relatively easy deployment options for server operators.

8 CONCLUSION AND FUTURE WORK

Passwords are likely to remain the de facto approach for authenticating users on the web, despite their inherent security weaknesses. Therefore, it is critical to improve the security of such systems without decreasing performance, usability, or deployability. A comprehensive solution must address the dual threats of phishing and theft of password databases, even in the case of rogue servers. We have demonstrated that SafeKeeper is a significant step towards meeting these objectives. As future work, we plan to implement selected extensions and variations, such as using SafeKeeper to protect email addresses from misuse, and integrating SafeKeeper into client-side password managers, as described in our technical report [24].

ACKNOWLEDGMENTS

This work was supported in part by the Intel Collaborative Research Institute for Secure Computing at Aalto University, and the Cloud Security Serivces (CloSer) project (3881/31/2016), funded by Tekes. M. Mannan is supported in part by an NSERC Discovery Grant and a NordSecMob Scholarship.

REFERENCES

[1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *International Workshop on Hardware and Architectural Support for Security and Privacy.* https://doi.org/10.1.1.405.8266

[2] APWG.org. 2016. Phishing Activity Trends Report (4th Quarter). (2016). http://docs.apwg.org/reports/apwg_trends_report_q4_2016.pdf.

[3] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. 2016. DROWN: Breaking TLS Using SSLv2. In *USENIX Security Symposium.* https://drownattack.com/drown-attack-paper.pdf

[4] Joseph Birr-Pixton. 2016. Using SGX to harden password hashing. (2016). https://jbp.io/2016/01/17/using-sgx-to-hash-passwords.

[5] A. Biryukov, D. Dinu, and D. Khovratovich. 2016. Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. In *IEEE European Symposium on Security and Privacy.* https://doi.org/10.1109/EuroSP.2016.31

[6] J. Blocki and A. Datta. 2016. CASH: A Cost Asymmetric Secure Hash Algorithm for Optimal Password Protection. In *IEEE Computer Security Foundations Symposium.* https://doi.org/10.1109/CSF.2016.33

[7] Joseph Bonneau. 2012. The Science of Guessing: Analyzing an Anonymized Corpus of 70 million Passwords. In *IEEE Symposium on Security and Privacy.* https://doi.org/10.1109/SP.2012.49

[8] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. 2012. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In *IEEE Symposium on Security and Privacy.* https://doi.org/10.1109/SP.2012.44

[9] Helena Brekalo, Raoul Strackx, and Frank Piessens. 2016. Mitigating Password Database Breaches with Intel SGX. In *Workshop on System Software for Trusted Execution.* https://doi.org/10.1145/3007788.3007789

[10] HTTrack Website Copier. 2017. (2017). https://www.httrack.com/.

[11] Qian Cui, Guy-Vincent Jourdan, Gregor V. Bochmann, Russell Couturier, and Iosif-Viorel Onut. 2017. Tracking Phishing Attacks Over Time. In *Conference on World Wide Web.* https://doi.org/10.1145/3038912.3052654

[12] Dan Cvrcek. 2014. Hardware Scrambling - No More Password Leaks. (2014). https://www.lightbluetouchpaper.org/2014/03/07/hardware-scrambling-no-more-password-leaks.

[13] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. 2014. The Tangled Web of Password Reuse. In *Network and Distributed Systems Symposium.* https://doi.org/10.14722/ndss.2014.23357

[14] The Breached Database Directory. 2017. (2017). https://vigilante.pw.

[15] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. 2015. The Pythia PRF Service. In *USENIX Security Symposium.* https://www.usenix.org/node/190917

[16] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. (2016). https://eprint.iacr.org/2016/204.

[17] Weili Han, Zhigong Li, Minyue Ni, Guofei Gu, and Wenyuan Xu. 2016. Shadow Attacks based on Password Reuses: A Quantitative Empirical View. *IEEE Transactions on Dependable and Secure Computing* (2016). https://doi.org/10.1109/TDSC.2016.2568187

[18] Kashmir Hill and Surya Mattu. 2017. Before You Hit 'Submit,' This Company Has Already Logged Your Personal Data (Gizmodo). (2017). https://gizmodo.com/before-you-hit-submit-this-company-has-already-logge-1795906081.

[19] Jun Ho Huh, Hyoungshick Kim, Swathi S.V.P. Rayala, Rakesh B. Bobba, and Konstantin Beznosov. 2017. I'm too Busy to Reset my LinkedIn Password: On the Effectiveness of Password Reset Emails. In *ACM SIGCHI Conference on Human Factors in Computing Systems.* https://doi.org/10.1145/3025453.3025788

[20] Intel Corporation. 2017. 1U System Delivering Cryptographic Isolation Technology. (2017). https://www.intel.com/content/www/us/en/data-center-blocks/business/secure-enclaves-blocks.html.

[21] Intel Corporation. 2017. Software Guard Extensions (Intel SGX). (2017). https://software.intel.com/en-us/sgx.

[22] David Jaeger, Chris Pelchen, Hendrik Graupner, Feng Cheng, and Christoph Meinel. 2016. Analysis of Publicly Leaked Credentials and the Long Story of Password (Re-)use. In *Conference on Passwords.* http://www.passwordresearch.com/papers/paper686.html

[23] Georgios Kontaxis, Elias Athanasopoulos, Georgios Portokalidis, and Angelos D. Keromytis. 2013. SAuth: Protecting User Accounts from Password Database Leaks. In *ACM Conference on Computer and Communications Security.* https://doi.org/10.1145/2508859.2516746

[24] Klaudia Krawiecka, Arseny Kurnikov, Andrew Paverd, Mohammad Mannan, and N. Asokan. 2017. Protecting Web Passwords from Rogue Servers using Trusted Execution Environments. (2017). https://arxiv.org/abs/1709.01261.

[25] Klaudia Krawiecka, Arseny Kurnikov, Andrew Paverd, Mohammad Mannan, and N. Asokan. 2018. SafeKeeper Project. (2018). https://github.com/safekeeper.

[26] Klaudia Krawiecka, Andrew Paverd, and N. Asokan. 2016. Protecting Password Databases Using Trusted Hardware. In *Workshop on System Software for Trusted Execution.* https://doi.org/10.1145/3007788.3007798

[27] Jake Leichtling. 2015. Continuing to protect Chrome users from malicious extensions. (2015). https://blog.chromium.org/2015/05/continuing-to-protect-chrome-users-from.html.

[28] PHP-CPP: A C++ library for developing PHP extensions. 2017. (2017). http://www.php-cpp.com/.

[29] Samuel Marchal, Kalle Saari, Nidhi Singh, and N. Asokan. 2016. Know Your Phish: Novel Techniques for Detecting Phishing Sites and Their Targets. In *IEEE International Conference on Distributed Computing Systems.* https://doi.org/10.1109/ICDCS.2016.10

[30] Netcraft.com. 2017. Let's Encrypt and Comodo issue thousands of certificates for phishing. (2017). https://news.netcraft.com/archives/2017/04/12/lets-encrypt-and-comodo-issue-thousands-of-certificates-for-phishing.html.

[31] PHPass: Portable PHP password hashing framework. 2017. (2017). http://www.openwall.com/phpass/.

[32] Paul Grassi and others. 2017. NIST Special Publication 800-63B, Digital Identity Guidelines, Authentication and Lifecycle Management. (2017). https://doi.org/10.6028/NIST.SP.800-63b.

[33] PhishTank.com. 2017. Statistics about phishing activity and PhishTank usage. (2017). https://www.phishtank.com/stats.php.

[34] Have I Been Pwned. 2017. (2017). https://haveibeenpwned.com/pwnedwebsites.

[35] Blake Ross, Collin Jackson, Nick Miyake, Dan Boneh, and John C. Mitchell. 2005. Stronger Password Authentication Using Browser Extensions. In *USENIX Security Symposium.* http://usenix.org/publications/library/proceedings/sec05/tech/full_papers/ross/ross.pdf

[36] Y. Sheffer, R. Holz, and P. Saint-Andre. 2015. RFC7457: Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). (2015). https://tools.ietf.org/html/rfc7457.

[37] W3Techs World Wide Web Technology Surveys. 2017. (2017). https://w3techs.com/.

[38] Santiago Torres and Justin Cappos. 2014. PolyPasswordHasher: Improving Password Storage Security. *;login: The USENIX Magazine* 39, 6 (Dec. 2014), 18–21. https://password-hashing.net/submissions/specs/PolyPassHash-v1.pdf.

[39] Chun Wang, Steve T.K. Jan, Hang Hu, and Gang Wang. 2017. Empirical Analysis of Password Reuse and Modification across Online Service. (2017). https://arxiv.org/abs/1706.01939v2.