PPSB: An Open and Flexible Platform for Privacy-Preserving Safe Browsing

Helei Cui, Yajin Zhou, Cong Wang, Xinyu Wang, Yuefeng Du, and Qian Wang

Abstract—Safe Browsing (SB) is an important security feature in modern web browsers to help detect new unsafe websites. Although useful, recent studies have pointed out that the widely adopted SB services, such as Google Safe Browsing and Microsoft SmartScreen, can raise privacy concerns since users' browsing history might be subject to unauthorized leakage to service providers. In this paper, we present a Privacy-Preserving Safe Browsing (PPSB) platform. It bridges the browser that uses the service and the third-party blacklist providers who provide unsafe URLs, with the guaranteed privacy of users and blacklist providers. Particularly, in PPSB, the actual URL to be checked, as well as its associated hashes or hash prefixes, never leave the browser in cleartext. This protects the user's browsing history from being directly leaked or indirectly inferred. Moreover, these lists of unsafe URLs, the most valuable asset for the blacklist providers, are always encrypted and kept private within our platform. Extensive evaluations using real datasets (with over 1 million unsafe URLs) demonstrate that our prototype can function as intended without sacrificing normal user experience, and block unsafe URLs at the millisecond level. All resources, including Chrome extension, Docker image, and source code, are available for public use.

Index Terms—Privacy preserving, safe browsing, web browser, malware, phishing.

1 Introduction

Safe Browsing (SB in short) is a popular security service adopted by modern web browsers, e.g., Chrome, Firefox, Safari, Edge, and Opera, to protect users against websites that attempt to distribute malware via *drive-by download* [1] or launch social engineering attacks via *phishing and deceptive content* [2]. Warning pages will be displayed to users when they try to "access dangerous websites or download dangerous files" [3]. Though SB service may be implemented in different ways, the general detection procedure (see Fig. 1) is to check if the URL to be visited is present on a list of unsafe URLs, collected and maintained by a remote server. As a side note, it is also common practice to reserve a local filter that contains either a whitelist [4] or a blacklist [5] on the client side to circumvent heavy communication overhead.

Although useful, when enabling such SB services, users do have privacy concerns that their visited URLs (i.e., browsing history) could be collected and further abused for various purposes, e.g., targeted advertisements, background scanning, and even government surveillance [6]. To this end, we conduct a survey of popular SB services regarding data they collect, and the result is summarized in Table 1. Surprisingly, except the Google Safe Browsing

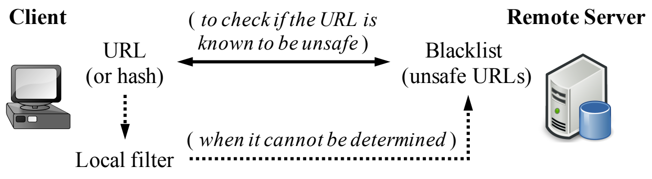

Client
Remote Server
(to check if the URL is known to be unsafe)
URL (or hash)
Blacklist (unsafe URLs)
Local filter
(when it cannot be determined)

Fig. 1. General procedure of Safe Browsing services.

(GSB) Update API [5], other services send URLs to remote servers, either in *full-length* hash value (i.e., Opera Fraud and Malware Protection [7]) or in the original form (e.g., Microsoft SmartScreen [4] and GSB Lookup API [8]). This suggests that the service providers (or other web browsers who adopt one of these SB services) could identify and obtain the URLs users visited.

By contrast, the GSB Update API (the current version is v4) [5] does not require the users' URLs nor the corresponding full-length hashes. Instead, the browser first canonicalizes the target URL and forms up to 30 decompositions (see some examples in Fig. 2). Next, it computes full-length hashes (via SHA-256) and truncates 4- to 32-byte hash prefixes for all expressions. A local database regarding all hash prefixes of unsafe URLs is reserved on the client side for filtering the truncated prefixes. Only when the decomposed target prefix matches any record in the local database will this prefix be transmitted to the server in order to derive a list of full hashes starting with the same prefixes for final matching. This seemingly privacy-friendly mechanism is now widely adopted in Chrome, Firefox [9], and Safari [10] with a massive user base, say over three billion devices use the GSB service [11].

However, recent research by Gerbet et al. [12] and its follow-up study [13] indicated that the underlying

- *H. Cui is with the School of Computer Science, Northwestern Polytechnical University, Xi'an 710129, China. E-mail: chl@nwpu.edu.cn*
- *Y. Zhou is with the School of Cyber Science and Technology, and the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China. E-mail: yajin_zhou@zju.edu.cn (Corresponding author)*
- *C. Wang, X. Wang, and Y. Du are with the Department of Computer Science, City University of Hong Kong, Hong Kong, China, and are also with the City University of Hong Kong Shenzhen Research Institute, Shenzhen 518057, China. E-mail: congwang@cityu.edu.hk, {xy.w, yf.du}@my.cityu.edu.hk*
- *Q. Wang is with the School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China, and is also with the State Key Laboratory of Cryptology, P.O. Box 5159, Beijing 100878, China. E-mail: qianwang@whu.edu.cn*

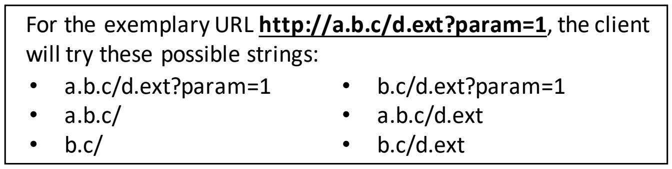
TABLE 1
A brief survey of data collected by popular Safe Browsing services.

SB Service	Data Collected	Known Products
Google Safe Browsing (Update API)	Hash prefix(es) of URL	Chrome[1], Safari, Firefox, Android WebView, etc.
Google Safe Browsing (Lookup API)	Full URL	(Experimental use)
Microsoft Windows Defender SmartScreen	Full URL	Windows, IE, Edge, and Chrome Extension
Opera Fraud and Malware Protection	Domain & hash of URL	Opera

anonymization technique of hashing and truncation fails when the server receives multiple prefixes for a URL. This kind of multiple prefix matching can reduce the uncertainty of URL re-identification or inference, due to the fact that the total number of URLs (and domains) on the Internet is finite and even sub-domain information might suffice for user tracking. What's worse, a malicious SB service provider can potentially abuse the GSB-like service for stealthy user behavior tracking. For instance, given a particular URL (e.g., relating to some political news), the SB provider could insert the hash prefixes of its (all) decompositions into the local filter. Later, once a user accesses the same URL (or the similar ones that share some decompositions, including the same domain), the matched prefixes would be sent to the remote server, and the SB provider can easily recover the visited URL (see Section 2 for a detailed description for this attack). The potential threat that the visited URLs could be recovered by a malicious SB service provider who applied the GSB Update API contradicts the privacy notice of Google Safe Browsing, i.e., the SB service provider "cannot determine the real URL from this information" [14]. Hence a Privacy-Preserving Safe Browsing (PPSB) service is desired to strengthen the protection of user privacy.

From another perspective, an integrated SB service, whose contents are contributed by different service providers, would certainly improve user experience as each provider is very likely to hold a list of unsafe URLs that others do not possess [12], [15], [16]. However, this also brings more challenges to our design given the following considerations. For starters, consider a service provider that owns a high-quality blacklist, which may be more frequently updated or simply contains more items. There is no incentive for this provider to publish its blacklist publicly or share it with other providers freely, as these are the most valuable asset of the blacklist providers. Directly sharing these blacklists with others in an uncontrollable way could make these dataset be obtained by every user, including the competitors. This is indeed the case in reality. For instance, a recent comparative study from NSSLab [17] suggests that the Microsoft SmartScreen [4] performs better than the GSB within the area of socially engineered malware protection, because Microsoft maintains a more updated blacklist. It

1. The latest Chrome 73 has an option "Help improve Safe Browsing", next to the "Safe Browsing" option. Once enabled, Chrome would periodically send extra system information and page content to Google.


For the exemplary URL **http://a.b.c/d.ext?param=1**, the client will try these possible strings:

- a.b.c/d.ext?param=1
- a.b.c/
- b.c/
- b.c/d.ext?param=1
- a.b.c/d.ext
- b.c/d.ext

Fig. 2. Exemplary decompositions for a queried URL.

is unlikely that Microsoft would share its blacklist in a transparent form so that other competitors could catch up. Besides, even if Microsoft is willing to share its blacklist in a protected manner, to the best of our knowledge, no existing mechanism whatsoever is able to support the integrated SB service whilst protecting the private blacklists. This tension between the competitive nature of different service providers[2] and the bright vision of a collaborative platform urges a novel solution that supports the integration of SB services provided by different parties without the exposure of providers' proprietary assets.

We now conclude that a PPSB service should satisfy the following formal requirements. First, the PPSB service should provide a strong privacy guarantee that the visited URLs cannot leave the user's browser in any form that could be re-identified by the service provider. Second, the PPSB service should provide a mechanism that allows the usage of blacklists from different providers, while at the same time, the privacy of these blacklists should be maintained. More precisely, the proprietary assets of the service providers cannot be revealed in any manner.

Our work. To meet the requirements, we design and implement an open and flexible platform for the PPSB service.

Compared with the GSB, to achieve the privacy guarantees, our design requires a relatively larger local database for storing the frequently updated encrypted blacklists, which are privately contributed by different providers with their own encryption keys. However, this brings challenges to our design. First, the client needs to search into the list of encrypted unsafe URLs without knowing its encryption key. To this end, we leverage the oblivious pseudorandom function (OPRF) [18] protocol to obtain a valid token with the aid of the key server maintained by the blacklist provider, and then check against the encrypted blacklist. During this process, though some information of the URL is needed to obtain the token from the key server, such information cannot be revealed thanks to the "oblivious" property of the OPRF protocol. Second, the encrypted blacklists should not bring too much storage burden on the clients. Meanwhile, as the valid unsafe URLs (or domains) change over time (e.g., the number of records in the constantly updated blacklists maintained by the famous blacklist providers PhishTank [15] and MalwareDomains [16] are usually less than 30,000), periodical update should be supported in a privacy-preserving manner. Thus, we further customize an efficient searchable encryption construction [19] into our design.

Besides that, our PPSB service retains the similar efficient processing flow to the GSB service [5]. In brief, the URL to

2. The blacklists from PhishTank [15] and MalwareDomains [16] may not need protection as they're publicly available. However, the mechanism of using extra blacklists at the same time is still missing in today's SB services.

be vetted is first converted into several expressions (e.g., the ones in Fig. 2) to prevent some potential unsafe URLs from slipping through the net. Next, the hashes of these expressions are computed via SHA-256, where the truncated hash prefixes of the vast majority of safe URLs can be quickly detected via a local filter. For those undetermined ones, the client needs to obtain OPRF tokens by sending the full-length *masked* hashes to a target key server, and then checks against the encrypted blacklist locally.

Unlike most existing SB services that primarily bind to a certain browser or use selected blacklists (either collected by themselves [3], [4], [20] or provided by cooperative blacklist providers [21]), our PPSB bridges the client applications (i.e., web browsers) and more blacklist providers. This boosts the capability of PPSB to detect unsafe URLs.

To make PPSB easier to use, we develop a client application, i.e., a Chrome extension. Hence, current Chrome users can directly use our PPSB service to detect unsafe URLs with privacy guarantees. Moreover, to encourage more third-party blacklist providers to join our PPSB platform with minimal efforts, we provide a fully functional API as well as a lightweight yet easy-to-deploy Docker image ($\approx$135 MB), so blacklist providers can focus on preparing high-quality and update-to-date blacklists (e.g., less than 3 MB for each encrypted blacklist built upon the frequently updated public data from the above-mentioned providers [15], [16]).

To the best of our knowledge, PPSB is the first design that enables *safe browsing with the guaranteed privacy protection* of users and blacklist providers simultaneously. The primary contributions are summarized as follows:

- We analyze existing SB services and give a summary of the potential leakage of these services. Inspired by [12], [13], we conduct thorough investigations upon active SB services, carry out concrete experiments upon users' privacy leakage, and report that the users' browsing histories could be leaked to (or inferred by) SB service providers, which raises privacy concerns to users.
- We propose the first PPSB service. It provides strong security guarantees that are missing in existing SB services. In particular, it inherits the capability of detecting unsafe URLs, while at the same time protects both the user's privacy (browsing history) and blacklist provider's proprietary assets (the list of unsafe URLs).
- We implement a full-fledged PPSB prototype, consisting of a client-side Chrome extension for users and a server-side API (and Docker image) for blacklist providers. The evaluation with real datasets and formal security analysis show the efficiency, effectiveness, and security strength of our design. All resources, including Chrome extension, Docker image, and source code, are available for public use[3].

2 Motivating Example

In this section, we will use an example to show why the state-of-art anonymization technique of the Google Update API still fails when adopted by a malicious service provider [12], [13], which motivates our work to propose a privacy-preserving SB service.

3. The PPSB Chrome extension: https://goo.gl/L2whw2, the Docker image on Docker Hub: https://hub.docker.com/r/ppsb/server/, and the code on GitHub: https://github.com/ppsb201804/PPSB.

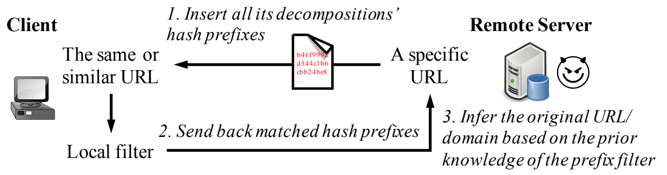

Client
The same or similar URL
Local filter
1. Insert all its decompositions' hash prefixes
b46f999a
d344c1b6
cbb24be8
...
2. Send back matched hash prefixes
A specific URL
Remote Server
3. Infer the original URL/ domain based on the prior knowledge of the prefix filter

Fig. 3. A possible way to infer users' browsing history by leveraging the low collision rate of hash prefixes of URLs.

Assume that a malicious SB service provider wants to know whether a user is visiting a particular web page, e.g., some political news. One way to achieve this is that the web browser sends all the visited URLs to a remote server, either in the plaintext, hash value or encrypted format. However, this behavior can be detected by monitoring and analyzing the browser, e.g., using the taint analysis technique [22]. A more stealthy way is the malicious SB service provider can identify the URLs users visited by manipulating the local hash prefix of the Google Update API.

Specifically, in order to track a particular URL (e.g., https://health.usnews.com/wellness/food/slideshows[4]), the SB service provider can insert the 32-bit hash prefixes of all its decompositions, e.g., `c01e362f` (health.usnews.com/wellness/food), `ae7b3778` (health.usnews.com/), `88a47746` (usnews.com/), etc., and then push this newly updated prefix filter to the clients. Later, once a user visits the web page (or similar URLs that share some decompositions), the matched hash prefixes would be sent to the remote SB server. Based on the prior knowledge of the prefix filter (i.e., the mappings between the hash prefixes and their corresponding URLs), the server can infer the URL (or domain) navigated by the user.

However, there is a chance that other URLs may have the same hash prefixes with the URLs tracked (hash prefix collision). In this case, the URL may be misidentified. To evaluate the possibility of hash prefix collision, Gerbet et al. [12] have calculated the number of unique URLs/domains that have the same hash prefix by using over 60 trillion URLs (which is the whole URLs collected by Google as of 2013). The results showed that as most 14,757 URLs and 3 domains can be located by using a single hash prefix. As the server receives multiple prefixes for each URL in the GSB, the uncertainty of URL/domain re-identification or inference can be reduced greatly, say as few as 2 prefixes are sufficient [12][5]. Thus, exposing (locally matched) 32-bit hash prefixes to the remote server can potentially be abused as a tool to track end users [12].

Note that, this way to track users is stealthy and hard to detect. First, only the hash prefixes of targeted users

4. We use this only as an example, and the potential attack is not limited to this URL.

5. We develop a tool to count the number of hash prefix collisions in a given URL list, inspired by [12]. Here, a collision means that a hash prefix is shared by two or more URLs. By using our dataset (that combines all unique URLs/domains from the Shallalist [23] dataset with over 1.7 million records and the Alexa-Top-1-Million-Sites [24] dataset with 1 million records), we observe that over 99.96% (i.e., 2,975,392) hash prefixes do not have any collisions (i.e., each of these 32-bit hash prefixes can uniquely point to its original URL decomposition) and all the remaining 0.04% (i.e., 1,118) hash prefixes only have 2 collisions (i.e., one prefix maps to two URLs). See GitHub: https://github.com/ppsb201804/PPSB/tree/master/experiments/test_gsb_prefix_attack

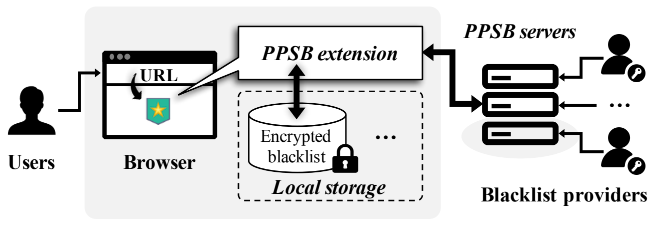


Fig. 4. Overview of our proposed PPSB service.

is manipulated. Other users are not affected. Second (and more important), users cannot decide if they have been tracked and which URLs are being tracked, since the exact URLs cannot be reverted solely via the local hash prefix.

3 Problem Statement

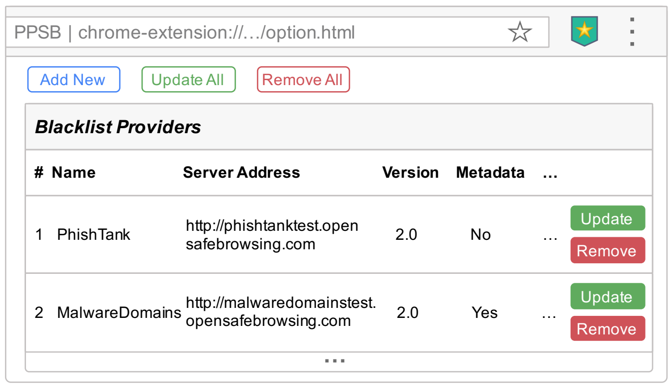
3.1 Overview

In order to detect unsafe URLs, existing SB services require the sharing of visited URLs, either in cleartext [4], [8], full-length hash [7] or 4- to 32-byte hash prefix [5], [25]. As a result, the SB service providers or adversaries could directly obtain users' browsing history, or indirectly infer the visited URLs by leveraging the shared information [12], [13]. This motivates our work to provide a privacy-preserving safe browsing (PPSB) service.

Our targeted PPSB service involves two different entities: the *user*, who uses a web browser to surf the Internet; the *blacklist provider* (or service provider), e.g., PhishTank [15], MalwareDomains [16], Netcraft [26], and Squidblacklist [27], who has expertise and capabilities to collect, verify and update a list of unsafe URLs. Hereinafter, we may interchangeably use "blacklist provider" and "service provider", because a blacklist provider can leverage our platform to provide PPSB services.

As shown in Fig. 4, users rely on a client application (i.e., the *PPSB extension* as a plug-in in existing web browsers) for checking whether the URL to be navigated is known to be malicious. Meanwhile, blacklist providers deploy their own *PPSB servers* for encrypting, publishing, and updating their collected URL blacklists and performing the designed PPSB service. Here, we choose browser extension as the form of our client application because it is minimally intrusive for users. For instance, the popularity of the Chrome SB extension released by Microsoft has shown great potential in this market (see Fig. 15 in Section 8).

After installing the PPSB extension, the user can add one or multiple blacklist provider(s) via an options page (see Fig. 5). Then the extension automatically downloads the latest database from the PPSB server of each specified blacklist provider, which includes an *encrypted blacklist* and a *hash prefix filter*. When performing the detection, the extension can quickly skip the vast majority of safe URLs by using the prefix filter without server interaction; for those potentially unsafe ones, the extension needs to interact with the PPSB server to obtain secure tokens (derived from the queried URL) and to perform final detection over the encrypted blacklist(s) locally, which is the major difference from the GSB. Unlike all existing SB services, the queried URLs and even hashes that can be used for prediction always remain private in the PPSB service.



Blacklist Providers

#	Name	Server Address	Version	Metadata	...	
1	PhishTank	http://phishtanktest.opensafebrowsing.com	2.0	No	...	Update Remove
2	MalwareDomains	http://malwaredomainstest.opensafebrowsing.com	2.0	Yes	...	Update Remove

...

Fig. 5. User interface of options page in PPSB extension.

3.2 Threat Model and Assumptions

Consistent with existing SB services, the blacklist providers have the incentive to collect and publish unsafe URLs for helping users to avoid websites that contain malware or phishing and deceptive content, e.g., for the better marketing purpose. We assume these blacklist providers and the corresponding PPSB servers are *semi-trusted*. They faithfully perform the designed procedures, i.e., the database preparation/update and the server-aided token generation. But they should not be aware of the queried URLs from users.

Meanwhile, we consider that users, even masqueraded by adversaries, should be unable to recover the collected unsafe URLs by using our PPSB extension in their web browsers, as these are the most valuable asset of the blacklist providers. Note that the correctly installed PPSB extension will not send the URLs or even the corresponding hashes in cleartext to any other parties. To ensure this, we publish the source code of the PPSB extension for public review.

Besides that, the potential URL leakage via the other channels (such as the aforementioned vulnerability when using GSB service, the defects or backdoors in browsers themselves, and other malicious extensions), and side-channel attacks based on power, traffic analysis or timing attacks (e.g., measuring the detection time) are out of scope.

Discussion on (Malicious) Blacklist Providers. As an open platform, a malicious party might leverage PPSB to degrade the client-side user experience, like inserting a number of fake or safe URLs or increasing the server-side delay. To address this potential issue, PPSB provides a flexible mechanism for users to add or remove blacklist providers (see Fig. 5). Moreover, some review-based mechanisms (e.g., reputation-based ranking) could be leveraged by our system to help users to choose the providers with good reputations, e.g., PhishTank. Though the malicious provider could bypass the review system to promote a (malicious) blacklist, the user's browsing history is still not leaked and thus being protected. Nevertheless, how to make the review system more robust is an open question, and the progress in this field could be borrowed by our system in the future.

3.3 Design Goals

• **Privacy-preserving:** to ensure that blacklist providers are unable to derive a user's URL from the information collected during the server-aided token generation process, and users are unable to recover the unsafe URLs from the downloaded database from a blacklist provider.

• **Fast processing:** to ensure that the PPSB service is introduced in a minimally intrusive way, i.e., the entire detection routine should be sufficiently fast.

• **Backward compatibility and easy customization:** to ensure that the PPSB service is compatible with existing popular web browsers, e.g., Chrome, and to allow users to add and delete other blacklist providers with minimal effort.

• **Fast deployment and auto synchronization:** to simplify the deployment effort of the PPSB server, and to allow blacklist providers to update and synchronize new versions to all clients in an easy and convenient way.

3.4 Preliminaries

Bloom Filter. Bloom filter is a space-efficient probabilistic data structure for high-speed set membership tests [28]. Briefly, it uses an m-bit array to represent a Set of at most n items. For each query item x, it will be mapped into k positions, which is calculated by a set of k independent uniform hash functions $\mathbf{H} = \{H_1, ..., H_k\}$. If $\bigwedge_{i=1}^{k} H_i(x) = 1$ ("$\bigwedge$" is the bitwise AND operation), x is probably in S (with a tunable probability ϵ of being wrong); otherwise, x is not in S (with no error). The lower the false positive rate ϵ is, the more bits the filter costs. Here, we use Bloom filter for prefix filtering, which allows a tolerable false positive rate.

Oblivious Pseudorandom Function (OPRF). OPRF protocol [18] enables two parties, say Alice holding an input message m and Bob holding a secret key k, to jointly and securely compute a pseudorandom function (PRF) $F(k, m)$, i.e., $\{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^l$. This two-party computation protocol is operated obliviously in the sense that Alice only learns the output value, while Bob learns nothing from the interaction. Here, we use an efficient OPRF instantiation based on the security of the *Elliptic Curve Discrete Logarithm Problem (ECDLP)*, namely EC-OPRF [18]: $F(k, m) = H_3(k \cdot H_2(m))$, where $H_2 : \{0,1\}^* \rightarrow E$ is a secure hash function that maps a binary string m of arbitrary length to a point G on an elliptic curve E, $H_3 : E \rightarrow \{0,1\}^l$ is a secure hash function that derives a binary string from an elliptic curve point $G \in E$, k is a randomly selected secret key, and $c \cdot G$ denotes the multiplication of a scalar integer c with a point G on an elliptic curve. In particular, Alice picks a random integer r and sends $X = r \cdot H_2(m)$ to Bob. Then Bob sends back $Y = k \cdot X$. Lastly, Alice computes $Z = r^{-1} \cdot Y = k \cdot H_2(m)$ and obtains $H_3(Z)$, i.e., $H_3(k \cdot H_2(m))$, as the PRF output. Note that Alice has access to the elliptic curve point $H_2(m)$ and $k \cdot H_2(m)$, but she is unable to recover Bob's secret k because this is the ECDLP and is known to be hard to compute [18].

4 The Proposed PPSB Design

4.1 Design Rationale

Almost every popular browser nowadays is equipped with a certain SB service to expand users' knowledge upon the safety of the website to be visited. The common practices adopted by existing SB services follow the natural design of directly processing users' transmitted data on the server side[6]. However, without any extra protection, data sent by users contains sufficient information that can be utilized on the server side to (re-)identify [12], [13] the users' browsing history. Therefore, it is essential that any data leaving the client side should be effectively masked such that the server cannot interpret any valuable information from the masked data. Meanwhile, to protect the proprietary knowledge of independent blacklist providers, some encryption method is indispensable here.

Given the above two considerations, the target problem can be formulated as follows. There are two parties, namely a blacklist provider and a user, who have a list of unsafe URLs $\mathbf{B}$ and a queried URL u, respectively. The user wants to verify whether or not $u \in \mathbf{B}$ while preserving $\mathbf{B}$ and u secret to each other. To solve this, we use the cryptographic primitive OPRF [18] to construct an *encrypted matching* scheme, similar to the scheme in [29]. In brief, the blacklist provider first generates each unsafe URL a secure token t via a PRF with a secret key k, and distributes the encrypted blacklist $\mathbf{D} = \{t_1, \cdots, t_n\}$ to clients for later detection via a server-aided encrypted matching scheme.

Data Structure for Encrypted Blacklist. Unlike the case of [29], the data structure of $\mathbf{D}$ is designed as the *Set* in our usage scenario. We intentionally choose not to use Bloom filter due to its inherent false positive rate, which is not suitable for the blacklist URL checking. Any false positive, even with a very small chance, would inevitably affect the user's surfing experience. This constraint compels us to adopt encrypted data structures that support exact URL testing in a private way. After obtaining a copy of $\mathbf{D}$, the client runs the OPRF protocol with the blacklist provider to jointly compute a valid token t_q without revealing their inputs, i.e., the secret key k of the provider and the queried URL u_q of the user. With t_q, the client can check if it is present on the blacklist $\mathbf{D}$ locally. If so, it would show a warning page immediately.

Supporting Metadata. Sometimes, the blacklist provider wants to provide extra metadata m corresponding to each unsafe URL u, which helps distinguish between threat types, e.g., *malware* or *phishing*. In this case, we consider the metadata should also be protected until a match is found since the metadata usually contains sensitive information, such as threat type, data source, and update time, which could incur potential side information leakage, like using frequency analysis to obtain some statistical information.

To this end, we further integrate the searchable symmetric encryption (SSE) construction [19] into the above fundamental design, i.e., using OPRF to generate the secure tokens used in SSE. So the above-mentioned secure token t will be associated with an encrypted value v derived from m, and m will be exposed *iff* a match t is found. The similar treatment has been successfully applied in many scenarios for obtaining a server-aided OPRF token while hiding the input and secret key from each other, e.g., secure data deduplication [30], [31], outsourced private information retrieval [32], similarity joins over encrypted data [33], and

6. A simple approach by directly sending a hash list of unsafe URLs and performing detection on the client is undesirable, as the shared blacklist can be abused stealthily and such proprietary knowledge would be violated.

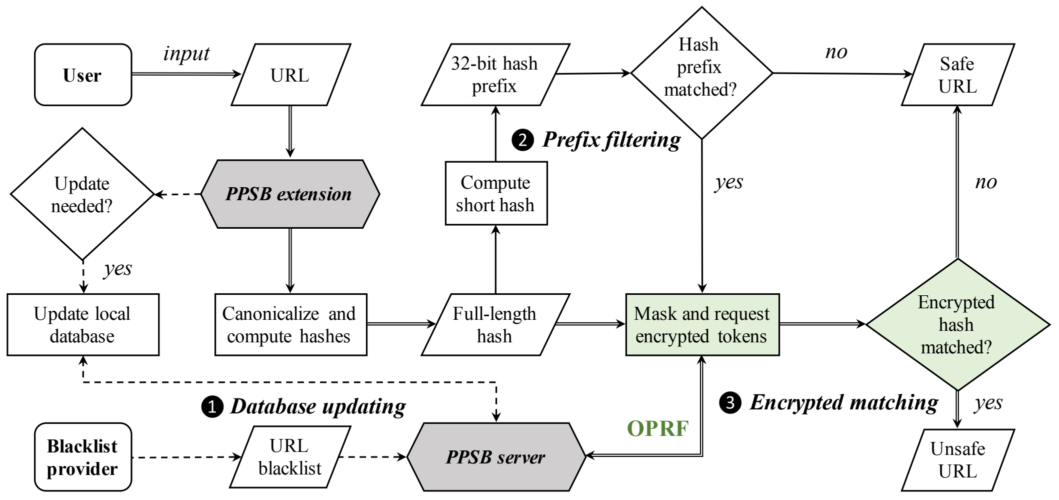


Fig. 6. Flowchart of our PPSB service: 1) the arrow $\dashrightarrow$ shows the subroutine of database updating initiated by a blacklist provider; 2) the arrow $\longrightarrow$ shows the subroutine of prefix filtering, which can quickly detect the vast majority of safe URLs; 3) the arrow $\Longrightarrow$ shows the main and complete detection routine when there is a match in the local prefix filter.

secure dynamic malware detection [34]. Back to our usage scenario, if metadata is deemed as an indispensable component to be shown, SSE essentially supports the output of metadata on the client side when matching the queried URL. In case the metadata is omitted by service providers, e.g., PhishTank would be a case where all types of dangerous URLs would be *phishing*, the standard of our usage scenario naturally degenerates, and a *Set* data structure combined with our server-aided OPRF token generation would serve the purpose.

Optimization via Prefix Filter. Following a similar approach adopted in the GSB Update API, we can also apply the prefix filtering technique to quickly detect the vast majority of safe URLs. To achieve this, the blacklist provider needs to publish an extra prefix filter that contains 32-bit (short) hash prefix of each unsafe URL on the blacklist. Due to the essential collisions incurred by this kind of short hash, others are unable to reliably discover the occurrence of a specific URL or recover all the unsafe URLs in the target blacklist offline. We emphasize that this is *completely different* from the vulnerable usage pointed in [12], [13], which leverages the correlated hash prefixes to re-identify the URL requested by a user. Because the data sent to the server is protected by the "oblivious" property of OPRF, which does not reveal the original hash prefix as in GSB.

Remark. We note that the design requirements of PPSB are also related to a line of prior art on private membership test [29], [35], [36]. However, those cryptographic designs either incur relatively higher communication delay (at the second level) [36], or heavier server-side operations [35], [36], both of which might impede the overall system scalability and *delay-sensitive* services that PPSB aims to achieve. Besides, as mentioned previously, the false positives inherited from Bloom filter in prior art [29], [35], [36] also do not quite fit into the safe browsing scenario. In the following, we elaborate the major steps of the PPSB service, also shown in Fig. 6. For completeness, we focus on the usage scenario that needs metadata to each unsafe URL, and the case without metadata can be simplified by omitting the metadata.

Algorithm 1 Build Encrypted Blacklist with Metadata

```
Input: The secret key of a blacklist provider: $\mathbf{K} = (k_1, k_2)$;
    the URL blacklist: $\mathbf{B} = \{(u_1, m_1), \cdots, (u_n, m_n)\}$, where
    $u_i$ is an unsafe URL and $m_i$ is its metadata.
Output: Encrypted blacklist $\mathbf{D}$.
 1: Initialize a dictionary (key-value) structure $\mathbf{D}$;
 2: for $i \leftarrow 1$ to $n$ do
 3:     $h_i = H_1(u_i)$;                          // $H_1 : \{0,1\}^* \to \{0,1\}^l$;
 4:     $G_i = H_2(h_i)$;                          // $H_2 : \{0,1\}^* \to E$;
 5:     $t_1 = F(k_1, h_i) = H_3(k_1 \cdot G_i)$;   // $H_3 : E \to \{0,1\}^l$;
 6:     $t_2 = F(k_2, h_i) = H_3(k_2 \cdot G_i)$;
 7:     $v_i = Enc(t_2, m_i)$;
 8:     $\mathbf{D}.put(t_1, v_i)$;
 9: end for
10: return $\mathbf{D}$;
```

4.2 Setup Stage: Database Building and Updating by Blacklist Provider

First of all, a blacklist provider should build an *encrypted blacklist* to compare with the queried URLs from users, without leaking the collected unsafe URLs. To achieve this, we apply one of the famous SSE constructions [19], which is implemented by using a generic dictionary (i.e., key-value store). Particularly, the blacklist contains a pair of encrypted key (unsafe URLs) and value (metadata of the corresponding URL, e.g., threat type, platform type, and cache duration). After finding a match in the blacklist, a warning page will be shown to users with the metadata.

Algorithm 1 illustrates the detailed procedure of building the encrypted blacklist $\mathbf{D}$ with metadata. We use u to denote the unsafe URL and m to denote the metadata. The objective is to transform the original URL blacklist (expressed in strings) $\mathbf{B} = \{(u_1, m_1), \cdots, (u_n, m_n)\}$ to a set of encrypted key-value pairs so that they can be stored in a dictionary $\mathbf{D}$. For each unsafe URL u_i, the blacklist provider first computes the full-length hash h_i via the hash function $H_1 : \{0,1\}^* \to \{0,1\}^l$ (i.e., SHA-256 in our prototype). Then the provider maps the hash value h_i to a point G_i on the elliptic curve E via the hash function

$H_2 : \{0,1\}^* \rightarrow E$. Next, a token pair (t_1, t_2) are generated from G_i: $t_1 = H_3(k_1 \cdot G_i)$ and $t_2 = H_3(k_2 \cdot G_i)$, where k_1 and k_2 are secret keys used for PRF $F(k,x) = H_3(k \cdot H_2(x))$ and $H_3 : E \rightarrow \{0,1\}^l$ is the hash function that maps an elliptic curve point $G \in E$ to a binary string. Lastly, the key-value pair (t_1, v_i) will be inserted into a dictionary $\mathbf{D}$, where v_i is encrypted metadata computed by using a symmetric encryption algorithm Enc with t_2 and m_i, i.e., $v_i = Enc(t_2, m_i)$. After this step, the encrypted blacklist $\mathbf{D}$ will be distributed to the users' browsers for PPSB service.

Keeping the Encrypted Blacklist Update-to-Date. To ensure protection against the latest threats, the encrypted blacklist $\mathbf{D}$ requires regular updates once the corresponding blacklist provider releases a new version. In practice, such an update does not always add records but may need to remove outdated ones, e.g., the number of records in the constantly updated blacklists maintained by PhishTank [15] and MalwareDomains [16] is usually less than 30,000. This is especially important in our design because we need to avoid any unnecessary client-side storage. Therefore, our design naturally supports dynamic update operations (i.e., *add* and *delete*) to the encrypted blacklist $\mathbf{D}$, while ensuring the forward privacy [37], [38], [39] (see more in Section 5). And these changes could be synchronized to users' local database periodically from the blacklist provider.

To add new records, one direct approach is to update the encrypted blacklist by inserting these newly prepared key-value pairs. Specifically, for each new record (u_{add}, m_{add}), the blacklist provider first computes $(t_1, Enc(t_2, m_{add}))$ as the same steps in Algorithm 1, where $t_1 = H_3(k_1 \cdot H_2(H_1(u_{add})))$ and $t_2 = H_3(k_2 \cdot H_2(H_1(u_{add})))$. Then the extension will fetch these encrypted key-value pairs and insert them into the corresponding $\mathbf{D}$. To delete those outdated records, the blacklist provider can compute the keys referring to the key-value pairs in $\mathbf{D}$, i.e., $t_{del} = H_3(k_1 \cdot H_2(H_1(u_{del})))$. Then the extension will remove them based on these keys accordingly.

Discussion on Metadata. The metadata in our design supports, in principle, a variety of information, since they can be expressed in *String*. Nevertheless, to ensure that the encrypted blacklist does not disclose the length of each metadata, the blacklist provider needs to use random paddings such that all metadata have a fixed length. If the metadata can be omitted, e.g., simply popping up a warning page to show the URL is unsafe, then the encrypted blacklist $\mathbf{D}$ can be stored in a *Set* or other space-efficient data structures with zero false positives, like *delta-encoded table* [12].

Remark. The first two real encrypted blacklists built from the datasets (14,583 records in PhishTank [15] and 27,013 records in MalwareDomains [16]) in our current prototype are only 1.11 MB and 2.79 MB, respectively, in uncompressed JSON format. Even when the number of records reaches the same scale as GSB (around 1 million unsafe URLs [40]), the size would be about 77 MB (see Fig. 14 in Section 7.3), which does not appear to be a real concern in modern desktop devices.

4.3 Offline Stage: Prefix Filtering by PPSB Extension

As mentioned before, the blacklist provider can prepare an additional prefix filter $\mathbf{F}$, which contains the 32-bit hash prefixes of unsafe URLs, for quickly detecting the vast majority of safe URLs. To obtain these hash prefixes, the blacklist provider can directly truncate each full-length hash h_i to a 32-bit short hash sh_i (after line 3 in Algorithm 1).

Once the prefix filter $\mathbf{F}$ is enabled, our PPSB service will perform a filtering operation, as an offline stage before actually conducting the detection over encrypted blacklist $\mathbf{D}$. Particularly, the PPSB extension checks if the hash prefix (sh_q) of queried URL is present in $\mathbf{F}$, realized via a Bloom filter. Similar to the GSB Update API, if no hash prefix is found, the URL is considered safe. Otherwise, the extension will continue to perform an online stage as follows.

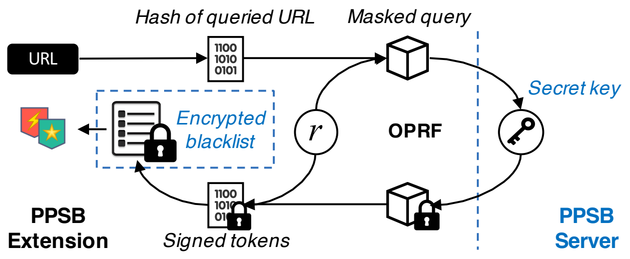


Fig. 7. The query flow of encrypted matching when there is a match in the local prefix filter.

4.4 Online Stage: Encrypted Matching by PPSB Extension and Server

From a high-level point of view, the major detection process is checking the URL to be visited by a user with the records in an encrypted blacklist. However, for the purpose of privacy protection, the URL should never be leaked to other parties (including the PPSB server and the blacklist provider) during the PPSB service. To this end, the PPSB extension generates tokens via an OPRF protocol with the help of the PPSB server and then tokens are checked over the encrypted blacklist locally. Once a match is found (i.e., the URL is unsafe), the corresponding web page will not be loaded and users will be warned with the threat information for further action. We follow the similar practice adopted by other built-in SB services, like the GSB [3] and the SmartScreen [4].

Fig. 7 and Algorithm 2 illustrate the detailed query flow and operations. Assuming the full-length hash h_q corresponding to a queried URL u_q has been computed before the above prefix filtering stage, the extension maps the h_q to a point G_q on the elliptic curve E via the same hash function H_2. Then it computes the masked point $X = r \cdot G_q$, where r is a one-time random value used for hiding the actual user's query from the PPSB server (maintained by a blacklist provider). Upon receiving X, the PPSB server generates the masked token pair (Y_1, Y_2) with its secret keys k_1 and k_2: $Y_1 = k_1 \cdot X$ and $Y_2 = k_2 \cdot X$, and returns them back to the extension. Next, the extension obtains the authorized token pair (t_1, t_2) by unmasking (Y_1, Y_2) with the same r and computing the hash value via H_3. At last, if t_1 matches a key in the $\mathbf{D}$, then the corresponding metadata m_q would be decrypted via $Dec(t_2, v_q)$, where v_q is obtained via $\mathbf{D}.get(t_1)$. Otherwise, the queried URL u_q does not refer to a known unsafe URL.

Algorithm 2 Detect Over Encrypted Blacklist with Metadata

```
Input: The secret key of a blacklist provider: $\mathbf{K} = (k_1, k_2)$;
    the encrypted blacklist: $\mathbf{D}$; and a suspect URL: $u_q$, which
    is present in the initial filter $\mathbf{F}$.
Output: A valid metadata $m_q$ or $\perp$.
    PPSB extension: // mask the hashed URL
 1: $h_q = H_1(u_q)$;                    // $H_1 : \{0,1\}^* \rightarrow \{0,1\}^l$;
 2: $G_q = H_2(h_q)$;                    // $H_2 : \{0,1\}^* \rightarrow E$;
 3: Pick a random integer $r \overset{\$}{\leftarrow} \mathbb{Z}$;
 4: $X = r \cdot G_q$;
 5: Send $X$ to the PPSB server;
    PPSB server: // generate tokens with the PRF keys
 6: $Y_1 = k_1 \cdot X$;   $Y_2 = k_2 \cdot X$;
 7: Send back $(Y_1, Y_2)$;
    PPSB extension: // unmask the query tokens and detect
 8: $t_1 = H_3(r^{-1} \cdot Y_1) = H_3(r^{-1} \cdot k_1 \cdot r \cdot G_q) = H_3(k_1 \cdot G_q)$;
 9: $t_2 = H_3(r^{-1} \cdot Y_2) = H_3(r^{-1} \cdot k_2 \cdot r \cdot G_q) = H_3(k_2 \cdot G_q)$;
10: if $\mathbf{D}.get(t_1) == null$ then
11:     return $\perp$, which represents the URL $u_q$ is safe;
12: else
13:     $v_q = \mathbf{D}.get(t_1)$;   $m_q = Dec(t_2, v_q)$;
14:     return metadata $m_q$ of the detected unsafe URL;
15: end if
```

Remark. The complete processing flow of our PPSB service is similar to the GSB Update API. Both involve prefix filtering and server-aided full-length hash comparison. The core difference is that, in PPSB, the blacklist is encrypted and pre-downloaded, and the data (i.e., hashes of queried URLs and signed tokens) are exchanged via an OPRF protocol.

5 Security Analysis

Our proposed PPSB service can achieve the following privacy guarantees: 1) From a user' perspective, the actual URL to be checked should never be leaked to the PPSB server (or the corresponding service/blacklist provider), as well as its associated hashes or prefixes, even when multiple queries (for different decompositions) are needed for a URL. This leads to stronger privacy protection of the user's browsing history. 2) From a blacklist provider's perspective, the encrypted blacklist should not leak the information of underlying unsafe URLs to the client (i.e., the extension), unless there is a match by using a token generated with the help of the server. This ensures that the information obtained by the clients is still the same as the existing SB services, even though the blacklist has been moved from the server to the client. Now, we analyze the specific designs.

Privacy in Prefix Filtering. We first consider the offline stage prefix filtering, which is used for quickly detecting the most of safe URLs and performed by the PPSB extension locally. The filter $\mathbf{F}$ itself only contains the 32-bit short hash of each unsafe URL. Thanks to the one-way property of the cryptographic hash function and the intrinsic collisions incurred by the short hash, a dishonest user can never use the prefix filter $\mathbf{F}$ to reliably recover all the unsafe URLs in the prefix filter offline.

Recall that a prefix match in $\mathbf{F}$ indicates the URL may be unsafe. Based on this, a dishonest user, who owns a blacklist, can infer a general picture of the target blacklist based on the intersection of the common prefixes. We admit that involving the prefix filter could, to some extent, leak the existence information of some known unsafe URLs. But this is not profitable for the users, because they cannot dig new knowledge regarding the unsafe URLs from the provider. And such (probabilistic) existence information is inevitably revealed to the client for safe browsing purpose.

Privacy in Encrypted Matching. We then focus on the encrypted matching stage as introduced in Section 4.4. Specifically, once a hash prefix is present in the local filter, the corresponding full-length hash needs to be securely converted into encrypted tokens by the server via an OPRF protocol, and then be checked against the local encrypted blacklist $\mathbf{D}$. Here, we follow the security notion of SSE [19], [41], [42] to justify that our encrypted matching scheme achieves security against adaptive chosen-keyword attacks under quantifiable leakage profiles. That is, the views of the client application (i.e., the PPSB extension) are formally defined in stateful leakage functions. Within a polynomial number of adaptive queries[7], the client only learns the information defined in leakage functions, no other information about the unmatched unsafe URLs in $\mathbf{D}$.

Precisely, three leakage functions are defined for the view of the encrypted blacklist, query pattern, and access pattern, where the query pattern indicates the equality of queried URLs and the access pattern includes the results of queries:

• Since the encrypted blacklist $\mathbf{D}$ is kept on the user's client, the extension knows its capacity and size, which are captured in the leakage function $\mathcal{L}_1$, defined as: $\mathcal{L}_1(\mathbf{D}) = (n, (|a|, |b|))$, where n is the number of records in $\mathbf{D}$, $|a|$ and $|b|$ are the bit lengths of encrypted key-value pairs.

• During the PPSB service, the extension sees the repeated queries, accessed key-value pairs, and matched metadata ms, which are captured in the query pattern $\mathcal{L}_2$, defined as: $\mathcal{L}_2(\{u_i\}_{1 \leq i \leq q}) = (\mathbf{N}_{q \times q})$, where $\mathbf{N}_{q \times q}$ is a symmetric binary matrix such that for $1 \leq i, j \leq q$, the element in the i-th row and j-th column is set to 1 if $u_i = u_j$, and 0 otherwise.

• Moreover, the extension also sees the matched results for the queries $\{u_i\}_{1 \leq i \leq q}$, which are captured in the access pattern, defined as: $\mathcal{L}_3(\{u_i\}_{1 \leq i \leq q}) = (\{(a, b)_i, m_i\}_{1 \leq i \leq q})$, where $(a, b)_i$ is the accessed key-value pair and m_i is the corresponding metadata.

Based on the above leakage functions, we can follow the security framework of [19], [41], [42] and give the simulation-based security definition of our encrypted matching scheme. It states that a probabilistic polynomial time simulator S can simulate an encrypted blacklist, respond a polynomial number of queries with simulated tokens and results, which are indistinguishable with the real encrypted blacklist, tokens, and results respectively.

***Definition* 1.** *Given our encrypted matching scheme* Π *with stateful leakage functions (*$\mathcal{L}_1$*,* $\mathcal{L}_2$*,* $\mathcal{L}_3$*), and a probabilistic polynomial time (PPT) adversary* A *and a PPT simulator* S*, we*

7. Using the rate-limiting strategy [30], e.g., up to 500 URLs per request in GSB Lookup API [8], can prevent an unlimited number of requests from a client for guessing the proprietary knowledge of the collected unsafe URLs.

define the probabilistic games $\mathbf{Real}_{\Pi,A}(\lambda)$ *and* $\mathbf{Ideal}_{\Pi,A,S}(\lambda)$ *as follows:*

- $\mathbf{Real}_{\Pi,A}(\lambda)$: *a challenger* C *generates secret keys* $\mathbf{K} = \{k_1, k_2\}$. *Then* A *selects a URL blacklist* $\mathbf{B} = \{(u_1, m_1), \cdots, (u_n, m_n)\}$ *and asks* C *to build the encrypted blacklist* $\mathbf{D}$ *via Algorithm 1. Then* A *adaptively conducts a polynomial number of secure queries with the tokens* $\mathbf{t} = \{(t_1, t_2), \cdots\}$ *generated from* C. *Finally,* A *returns a bit "1" as the game's output if the detection results are all correct and consistent; otherwise, "0".*
- $\mathbf{Ideal}_{\Pi,A,S}(\lambda)$: A *selects* $\mathbf{B}$, *and* S *generates* $\widetilde{\mathbf{D}}$ *based on* $\mathcal{L}_1$. *Then* A *adaptively conducts a polynomial number of secure queries. From* $\mathcal{L}_2$ *and* $\mathcal{L}_3$ *of each query,* S *generates the corresponding* $\widetilde{\mathbf{t}}$, *which are processed over* $\widetilde{\mathbf{D}}$. *Finally,* A *returns a bit "1" as the game's output if the simulated results are all correct and consistent; otherwise, "0".*

Our encrypted matching scheme Π *is* $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3)$*-secure against adaptive chosen-keyword attacks if for all PPT adversaries* A, *there exists a PPT simulator* S *such that*

$$Pr[\mathbf{Real}_{\Pi,A}(\lambda) = 1] - Pr[\mathbf{Ideal}_{\Pi,A,S}(\lambda) = 1] \leq negl(\lambda),$$

where $negl(\lambda)$ *is a negligible function in* λ.

The proof is given as follows, which demonstrates that the client-side PPSB extension only knows the above precisely defined leakage for a set of adaptive queries, and no other information else.

***Theorem* 1.** *Our encrypted matching scheme* Π *is* $(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3)$*-secure against adaptive chosen-keyword attacks in the random oracle model if* (Enc, Dec) *is semantically secure, and* F *is secure PRF.*

Proof. Based on $\mathcal{L}_1$, the simulator S can build a randomized blacklist $\widetilde{\mathbf{D}}$ with the same size as the real $\mathbf{D}$, containing n key-value pairs. Each simulated key-value pair $(\widetilde{a}, \widetilde{b})$ are random strings with the same lengths as the real encrypted key-value pair (a, b). Thanks to the semantic security of symmetric encryption and the pseudorandomness of secure PRF, $\widetilde{\mathbf{D}}$ is computationally indistinguishable from $\mathbf{D}$.

When the first query u_1 is sent, S randomly generates two strings as the simulated tokens $(\widetilde{t_1}, \widetilde{t_2})$. After that, a random oracle $\mathcal{O}_1$ is operated in the way of $\widetilde{a} = \mathcal{O}_1(\widetilde{t_1})$ to select $(\widetilde{a}, \widetilde{b})$. Then the other random oracle $\mathcal{O}_2$ is operated to get the result (i.e., metadata) $m = \mathcal{O}_2(\widetilde{t_2}||\widetilde{b})$. Note that m is exactly the same as the real result indicated in $\mathcal{L}_3$. And the due to the pseudorandomness of secure PRF, $(\widetilde{t_1}, \widetilde{t_2})$ are also computationally indistinguishable from (t_1, t_2). In the subsequent queries, the repeated tokens are recorded by $\mathcal{L}_2$, so S can directly use the previously simulated ones. Otherwise, S generates tokens via $\mathcal{O}_1$ and $\mathcal{O}_2$ in the same way as mentioned above. And the results derived from $\widetilde{\mathbf{D}}$ are also identical to the real results. Therefore, A cannot differentiate the simulated tokens and results from the real tokens and results. □

Discussion on Forward Privacy. The recent advancement in dynamic searchable encryption considers a stronger security setting, namely *Forward Privacy* [37], [38], [39]. Under this setting, the newly updated records cannot be related to previous search results. Otherwise, an adversary might be able to launch file-injection attacks to recover the queried record [43].

Our PPSB design can naturally ensure this forward privacy, i.e., the extension installed in the users' browsers is unable to learn the fact that the updated blacklist records match some records they previously checked. This is because all the records in the encrypted blacklist $\mathbf{D}$ are distinct from each other, including the initial ones and subsequently added ones. In other words, the newly added key-value pairs (or the "key" tokens when metadata is omitted) have no connection with the previously checked ones.

In addition, the authors of [39] also described the *Backward Privacy*, where (fresh) search queries should not leak matching records that have been deleted before. To our best knowledge, ensuring this privacy incurs extra overhead in either computation or communication [38], [39], and we will explore if there is a more efficient scheme that can fit in our *delay-sensitive* service. Nevertheless, the blacklist provider can periodically update its secret key when performing a full update (which excludes a batch of outdated records). This, to some extent, limits the update leakage in the backward privacy setting, as well as the forward privacy setting, even when a certain URL was added and deleted repeatedly. This is also supported by our current design as the key is always maintained by the provider.

Discussion on the Case Without Metadata. In case the metadata is omitted, the security strength can still hold because the encrypted blacklist $\mathbf{D}$ can be viewed as a *Set* that only contains "keys" rather than the key-value pairs as discussed above. Thus, the leakage function $\mathcal{L}_1$ will not cover the bit length of the "value" (i.e., $|b|$), and $\mathcal{L}_3$ will become $(\{a_i, o_i\}_{1 \leq i \leq q})$, where o_i is the 1-bit detection result (safe or unsafe). Here, we skip the detailed analysis of this use case as it is similar to the above case with metadata.

6 Implementation

6.1 Client Application: A Chrome Extension

To make PPSB easier to use and comparable with GSB on the same stage, we build our client application in the form of Chrome extension. The PPSB extension aims to block the connection request to a potential hazardous URL while preserving the privacy of users' browsing history. Similar to the GSB, our PPSB extension also requires local storage for data well prepared by blacklist providers.

6.1.1 Local Storage

As one of the key components in our PPSB design, the local storage mainly includes two data structures, i.e., the prefix filter $\mathbf{F}$ and the encrypted blacklist $\mathbf{D}$. All data with respect to these two data structures is fetched from some PPSB server(s) maintained by corresponding blacklist provider(s). Note that for a massive blacklist on the same scale as GSB (around 1 million unsafe URLs [40]), the uncompressed encrypted version in JSON format would occupy around 77 MB local storage on the client side. Our implementation carries through the principle of accelerating the client-side operations as much as possible, achieving a rather low-level runtime memory cost in the meanwhile.

- **Prefix Filter.** As mentioned in Section 4.3, the prefix filter is applied to avoid frequent communication over the

TABLE 2
Performance comparison among three data structures that can be used for prefix filter (#record = 50,000).

Candidate	Avg. Query Time and Standard Deviation (μs)	Memory (MB)
Bloom filter [44]	0.722 ($\pm$ 0.170)	1.866
Delta-encoded table [45]	145.5 ($\pm$ 94.06)	1.778
Set [46]	0.201 ($\pm$ 0.105)	3.057

network and consequentially decrease response time dramatically. In principle, the prefix filter can be realized via any data structures that support membership tests, such as the standard *Set* in JavaScript, *Bloom filter*[8] [44], and *delta-encoded table* [12]. To evaluate which one performs best in our usage scenario, we evaluate the membership query time and memory footprint of them in the browser JavaScript environment. For setup, we generate 50,000 random yet uniformly distributed integer strings as the input collection for the three data structures. Particularly, we order these inputs and divide them into five groups to obtain statistic variance apart from the average query time over 50,000 independent experiments.

Table 2 shows the performance comparison among the three data structures that can be used for prefix filter. Both Set and Bloom filter exhibit stable query speed at the sub-microsecond level in the course of our experiments while delta-encoded table is hundreds of time slower and exhibits significant variability in query speed as a consequence of query-time recovery from the encoded delta in essence. Moreover, we measure the heap used with the aid of V8 engine built-in library and it turns out the memory edge of delta-encoded table is insignificant and difficult to optimize in this high-performance JavaScript engine. From this, we prefer Bloom filter in our prototype on account of the balanced performance of query speed and memory footprint.

• **Encrypted Blacklist.** As introduced in Section 4.4, a local copy of each encrypted blacklist $\mathbf{D}$ is reserved on each user's client to facilitate the encrypted matching operations. Yet its implementation is rather straightforward. Fundamentally a generic dictionary (i.e., *Map* in JavaScript) is maintained to keep track of the encrypted key-value pairs containing encrypted token (derived from the hash of each unsafe URL) and the corresponding metadata information. In our current prototype, the metadata consists of the threat type (e.g., malware, phishing, and others) and the blacklist provider's ID, represented as the integer type for compression. Also, if the metadata is omitted, the *Set* in JavaScript can hold the encrypted tokens. Here, we do not consider the other space-efficient data structure, like the above-mentioned Bloom filter and delta-encoded table, because they either incur false positives or perform less efficiently.

Handling Frequent Update. To ensure protection against the latest threats, both $\mathbf{F}$ and $\mathbf{D}$ require regular updates once the corresponding blacklist providers release the latest versions. Note that every update comprises not only the *add* operation associated with recently incorporated unsafe URLs, but also the *delete* operation accountable for the removal of outdated items. It is necessary that obsolete information in $\mathbf{D}$ is removed timely by corresponding blacklist providers in order to optimize client-side storage. Note that our design naturally supports the two update operations owing to the distinctness of the key-value pairs in $\mathbf{D}$ (see Section 4.2). Besides, we use the standard Bloom filter as $\mathbf{F}$ in our current prototype, which does not support the "delete" operation and requires a full update. This is sufficiently efficient as the size of hash prefixes in $\mathbf{F}$ is relatively small, say $\approx$1.86 MB for 50,000 hash prefixes loaded in self-managed V8 JavaScript engine. We are aware that the partial update mode can also be made available by using some "dynamic" Bloom filter at the cost of additional overhead [47].

8. Unlike the exact detection scenario, the Bloom filter can be used here as it is meant for filtering and consequentially allows tolerable false positives.

6.1.2 Browser Extension

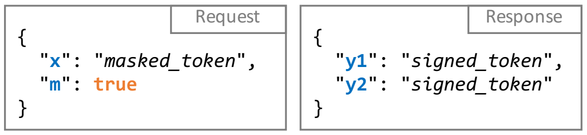
With the Chrome APIs, our extension is able to intercept the connection request and check locally or interactively with the PPSB server(s). For ease of exposition, we describe the implementation details following client-side procedures.

• **Adding Blacklist Provider(s).** Once the user installs the PPSB extension, a default blacklist source is included, which is maintained by ourselves. For a quick switch to other blacklist providers, we provide an options page as shown in Fig. 5. The user only needs to fill in an address of the corresponding PPSB server, which may be published on the provider's website. Any operation of add/delete of blacklist provider(s) triggers an auto download/removal of affected $\mathbf{D}$ and $\mathbf{F}$.

• **Pre-processing of URLs.** As a starting point of detection, each valid URL that follows RFC 2396 [48] is canonicalized into a normalized format. Following the GSB Update API, we use different combinations of the host suffix and path prefix, as shown in Fig. 2, and treat each of these expressions independently in the following procedure to prevent some potential dangerous URLs from slipping through the net. Afterwards, the PPSB extension computes full-length SHA-256 hashes and further truncated 32-bit hash prefixes, which are used for prefix filtering and encrypted matching operations respectively.

• **Server-Aided Token Generation.** Once a hash prefix sh is present in $\mathbf{F}$, then its full-length hash h will be securely transformed into encrypted tokens. Particularly, an interaction with the PPSB server via the asynchronous WebSocket call is performed to wrap up the OPRF protocol. In our prototype, we implement EC-OPRF [18] on top of Stanford JavaScript Crypto Library (SJCL) [49], which outperforms other OPRF instantiations (e.g., RSA-OPRF [30]) in terms of computation and bandwidth. At the end of the interaction, the extension obtains the authorized tokens (t_1, t_2) for the later encrypted matching process. The actual HTTP request/response body is shown in Fig. 8.

• **Final Detection.** Adhered to the description in Algorithm 2, line 10 to 15, once corresponding metadata is derived from the encrypted blacklist $\mathbf{D}$, (i.e., the queried URL is unsafe), the extension will redirect the current tab to a warning page with information like the threat type and the data source. If none such metadata is ever returned, the web page will be loaded as usual.



Request
```
{
  "x": "masked_token",
  "m": true
}
```

Response
```
{
  "y1": "signed_token",
  "y2": "signed_token"
}
```

Fig. 8. The request/response body in the server-aided OPRF token generation in JSON. Here, "m" is *true*, indicating the encrypted blacklist has the metadata field.

Optimization via Parallel Processing. To speed up the detection procedure in the case of multiple blacklists, we leverage the asynchronous requests of Chrome to process the above detection operations (except the URL pre-processing) in parallel. Once a negative result (i.e., the URL is known unsafe) is obtained, the detection procedure would be stopped and the warning page would be popped up immediately. Our testing results show that the extra overhead is modest when adding more blacklists, see Table 3 in Section 7.2.

6.2 PPSB Server for Blacklist Providers

In our design, each blacklist provider maintains a PPSB server, responsible for distribution of the encrypted blacklist and token generation for the clients. Below introduces the core APIs and a Docker-aided deployable prototype.

6.2.1 *Exported API*

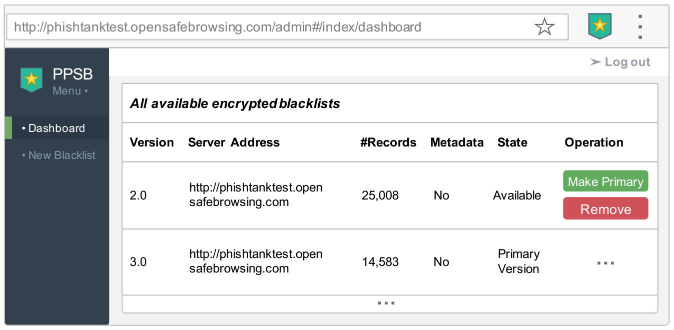
The server-side API enables a blacklist provider to deploy its own PPSB service. In the current prototype, we expose a fully functional API in JavaScript, which contains the following three major functions:

• $\mathbf{K} \leftarrow GenKey()$. It generates a pair of new secret keys $\mathbf{K} = (k_1, k_2)$, which are used for building the encrypted blacklist and generating the OPRF tokens.

• $\mathbf{D} \leftarrow BuildEncryptedBlacklist(\mathbf{K}, \mathbf{B}, meta)$. It outputs the encrypted blacklist $\mathbf{D}$, which contains a collection of encrypted key-value pairs, as shown in Algorithm 1. In particular, the blacklist provider should prepare the URL blacklist $\mathbf{B}$ in JSON format, e.g., [{"u": "url_string", "m": $metadata\_string$} . . .]. To keep the encrypted blacklist sufficiently concise, the metadata m in our current prototype only contains the threat type. Note that the other informative metadata, such as threat platform, attacking target, and update time, could be involved in our future version, since they can always be expressed in *Strings*. Here, the last parameter $meta$ (a Boolean value) indicates if the metadata can be omitted.

• $(Y_1, Y_2) \leftarrow SignToken(\mathbf{K}, X)$. It takes the secret key $\mathbf{K} = (k_1, k_2)$ and the masked token X as inputs, and outputs the signed tokens (Y_1, Y_2) which can be unmasked by the PPSB extension for later detection over the encrypted blacklist. Note that, the masked token X is sent by the client application, and the actual URL cannot be recovered (or inferred) from this masked token, which is guaranteed by the "oblivious" property of OPRF.

Apart from these, other functions like hash prefix extraction and data synchronization are also provided. Due to the constrained space, we omit them here and more detailed documentation could be found on GitHub.



http://phishtanktest.opensafebrowsing.com/admin#/index/dashboard
PPSB
Menu
• Dashboard
• New Blacklist
Log out
Make Primary
Remove

All available encrypted blacklists

Version	Server Address	#Records	Metadata	State	Operation
2.0	http://phishtanktest.open safebrowsing.com	25,008	No	Available	Make Primary Remove
3.0	http://phishtanktest.open safebrowsing.com	14,583	No	Primary Version	...
...					

Fig. 9. Web interface of PPSB server-side management.

6.2.2 *Fast Deployment via Docker*

To minimize the efforts for the blacklist providers to deploy the PPSB service, we release a lightweight and stand-alone *Docker image* to Docker Hub. Specifically, we use JavaScript inside the Docker and leverage the SJCL library [49] with the NIST P-256 elliptic curve to implement the EC-OPRF protocol. This docker image also provides a Web-based control panel to facilitate the operations of upload, encryption, and publication, as shown in Fig. 9. For simplicity, the involved cryptography configuration is fixed in our current released Docker image, i.e., updating the parameters needs to update the Docker image. To make this update more flexible, we plan to provide an interface to enable the operator to change such parameters dynamically and synchronize the new setting to end-users automatically.

To further improve the scalability when handing a huge number of requests, we also provide another *cluster setup*. It involves an extra load balancer (Nginx [50] or HAProxy [51]) to manage multiple PPSB backend services, automatically.

6.3 Limitations in Current Prototype

Though the design principle and workflow of our proposed PPSB service are independent of particular web browsers, the current APIs for client applications are in the JavaScript language and the current prototype is in the form of the Chrome extension, which does not support mobile devices. As future work, we will implement the APIs in more languages, e.g., the Go APIs or the Python APIs, and explore the possibility to port the PPSB to mobile platforms, e.g., Android and iOS. However, implementing PPSB on mobile devices, compared to desktop computers, faces different engineering efforts and new challenges. Our goal is to introduce PPSB in minimally-intrusive fashion. Accordingly, we use a browser extension for desktop computers. However, for mobile browsers, major vendors currently do not support third-party extensions. Thus, it would require building another native browser app from scratch. It is in our future roadmap, and we are yet to address the energy cost, limited storage, and other mobile constraints. With that said, we believe our desktop PPSB does have value. Firstly, desktop traffic still amounts to half of all browser traffic today [52]. Secondly, our design builds solid foundations for mobile PPSB platform, a necessary next step.

Meanwhile, we would explore other recently popular approaches like homomorphic encryption [53], [54] and hardware enclaves (i.e., Intel SGX) [55], [56] for supporting even more advanced computation over encrypted data while maintaining the efficiency.

Besides that, in the current starting stage, we only include two blacklist providers, i.e., PhishTank [15] and MalwareDomains [16], and the servers are maintained by ourselves. In the near future, we hope more blacklist providers would join our platform and provide more update-to-date and platform-specific[9] blacklists, so as to boost the capability of safe browsing.

7 Experimental Evaluation

To see the performance overhead of our PPSB prototype, we conduct the following experiments to answer the two questions: *First, what is the overhead of our introduced operations, e.g., the prefix filtering and the query of the encrypted database when the prefix is matched, to the client application (i.e., the Chrome extensions)? Second, what is the overhead of our system to the blacklist server, since it needs to generate the OPRF tokens, and periodically build the encrypted database of unsafe URLs?*

7.1 Setup

First of all, we deploy three PPSB servers acting as the role of three independent blacklist providers on three AWS EC2 instances "c5.large" (2 vCPU with 4 GB RAM) in Linux (Ubuntu Server 16.04 LTS). Precisely, three real datasets are used respectively in each instance: B1 - *PhishTank* [15] that contains 14,583 verified unsafe URLs; B2 - *MalwareDomains* [16] that contains 27,013 malware domains; and B3 - *Shallalist* [23] that contains 1,571,617 unsafe domains[10]. Hereinafter, we will not differentiate between "URL" and "domain" as they both are expressed by way of a string.

Second, the client-side performance is evaluated on a MacBook Pro (2.6 GHz Intel Core i7 with 16 GB RAM). Here, we study three scenarios in terms of the number of blacklist providers added in the PPSB extension. For comparison, we also test the latest Chrome 73 (official build 64-bit) with the built-in GSB service[11] and the recently released Chrome extension by Microsoft, namely *Windows Defender Browser Protection* (abbr. MSB) [20]. Without further elaboration, 10 individual experiments are performed to derive the average results in the following figures and tables.

Last but not least, the communication between PPSB extension and server (e.g., the encrypted blacklist synchronization and server-aided token generation) is reliably authenticated by using HTTPS.

7.2 Client Application

To answer the first question, we measure the latency when accessing unsafe and safe URLs with our PPSB extension.

9. Aware that threats are sometimes posed to a specific platform, e.g., Windows, macOS, and Linux, we suggest subdividing the blacklist according to the targeted platforms as this can further reduce the memory footprint of the local database.

10. The Shallalist has over 1.7 million domains and URLs, some of which are outdated or invalid. Here, we just use 1.5 million domains to evaluate the performance when PPSB faces a big dataset (e.g., the same scale as GSB).

11. To disable the GSB service in Chrome 73, users need to switch off the "Safe Browsing" option under the "Privacy and security" settings.

TABLE 3
Average load time of random unsafe URLs for three safe browsing services on major platforms.

Platform	GSB (ms)	MSB (ms)	PPSB w/ **B1** (ms)	PPSB w/ **B1,2** (ms)	PPSB w/ **B1,2,3** (ms)
Windows	112.6	116.8	333.5	388.1	437.7
macOS	184.7	194.3	340.5	373.3	440.1
Ubuntu	67.6	155.4	329.7	354.3	431.1

Note: B1 - PhishTank, B2 - MalwareDomains, B3 - Shallalist.

TABLE 4
Average load time of random unsafe URLs when handling three user requests simultaneously.

#Users	PPSB w/ **B1** (ms)	PPSB w/ **B1,2** (ms)	PPSB w/ **B1,2,3** (ms)
1	340.5	373.3	440.1
2	346.1	382.6	448.2
3	353.9	388.5	455.4

Note: B1 - PhishTank, B2 - MalwareDomains, B3 - Shallalist.

Comparable Speed with Other SB Services. First, we measure the initial load time of the prefix filter and encrypted blacklist from the default blacklist source (i.e., MalwareDomains, which can also be removed by the user). The average load time is about 750 ms, which is a one-time cost for each update of the default source. Meanwhile, we want to point out that our PPSB extension does not incur a noticeable delay when launching Chrome, just around 67 ms latency.

Moreover, to measure the load time of warning pages in case of accessing unsafe URLs, we randomly select 10 URLs that can be blocked by all the three SB services (i.e., GSB, MSB, PPSB), and test them on the latest stable version of Windows 10, macOS High Sierra, and Ubuntu 16.04. We use the following method to measure the load time with the help of Chrome exported APIs. Specifically, we mark the start time when Chrome is about to start a *navigation event*, and obtain the end time with the help of updated tab information. We then calculate the interval between these two Chrome built-in events as the load time of the unsafe URL. We take a similar method to calculate the load time of a safe URL.

Table 3 illustrates the average load time for different SB services on major OS platforms to load unsafe URLs (and being blocked). Due to the extra introduced operations (see the left part of Table 5), our PPSB prototype exhibits consistent overhead on all platforms. However, the load time is still at the millisecond level (less than 500 ms) and completely acceptable and doubtlessly unnoticeable[12] compared with the average page load times in 2018 (at the second level) [57]. And owing to the parallel computing techniques used on the client side, the scenario of PPSB with all the three providers incurs about 100 ms extra time cost on the basis of PPSB with one provider. This suggests the

12. Users are welcome to send us online feedback via https://goo.gl/forms/0I2KZX88cRv6AgJA3. As of now, we have received feedback from over 35 volunteers in our university, where the summary can be found at https://goo.gl/hFjL7o.

TABLE 5
Evaluation of the major operations in server-aided token generation.

PPSB Extension			PPSB Server
PrefixFilter (μs)	*Mask* (ms)	*Unmask* (ms)	*SignToken* (ms)
0.722	62.51	27.76	125.28 (w/ metadata) 58.36 (w/o metadata)

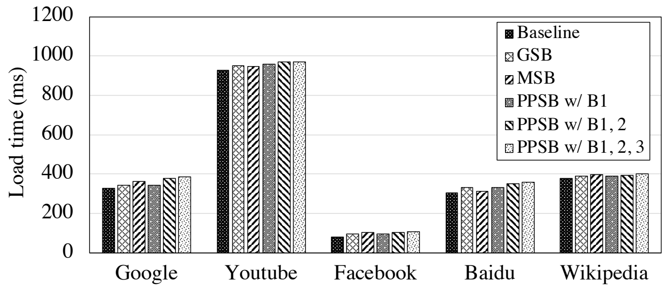


Fig. 10. Evaluation of the average load time of five famous (safe) websites (100 individual tests).

gentle linear growth of block time as the number of added blacklist providers increases gradually on the client side, which is an inevitable cost for the benefit of obtaining more comprehensive information from multiple sources. Besides that, Table 4 shows the average load time when three users access different unsafe URLs simultaneously and only a PPSB Docker instance handles all of them. We can observe that the average load time for each end user is not increased significantly thanks to the server-side parallel processing (see more discussion in Section 7.3).

We further evaluate the load time of safe URLs. Fig. 10 shows the load time of the five most frequently visited websites [58] in a real network environment by using a new anonymous Chrome test account with cache cleared. Here, the "Baseline" indicates the case that all SB services are disabled. As is evident from the results, similar to the GSB and MSB, our PPSB extension only introduces little overhead (e.g., 11 ms to 55 ms) in all three usage scenarios thanks to the fast-processing of local filter (about 0.722 μs for each query) for the vast majority of safe websites. Note that YouTube, as a typical content-heavy website (with more data to be loaded), needs a longer time than others, and Facebook is even faster than search engines as it only displays the login page in our test.

In summary, our PPSB prototype does introduce extra overhead as expected. However, the overhead of load time of unsafe URLs is within the millisecond level, and that of safe URL is unnoticeable in contrast with the dominant factors like network fluctuations.

7.3 PPSB Server

The PPSB server needs to build the encrypted blacklist and generate the OPRF tokens. Now, we evaluate the performance overhead caused by these two operations.

Server-Aided OPRF Token Generation. When there is a match in the local prefix database, the client application will generate a mask token and sends it to the server. The server then generates the OPRF token and returns it to the client for looking up in the encrypted blacklist. Since this operation is computation-intensive, it is the most time-consuming operation on the server side. Table 5 also shows the time used to generate the token(s) for each query in accordance with the existence of metadata.

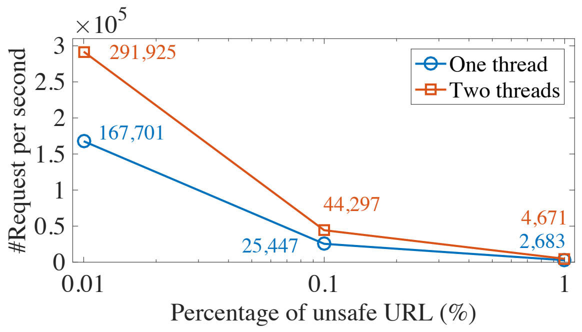


Fig. 11. Estimation of the number of SB requests per second that our PPSB server can support.

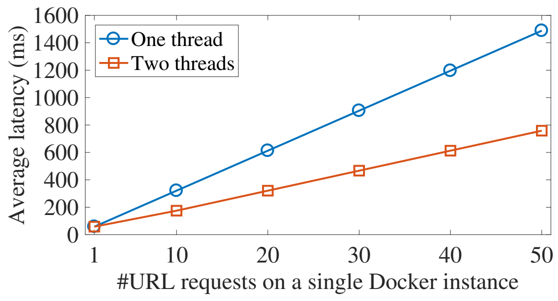


Fig. 12. Estimation of the average latency for a client getting the authorized token under the influence of increasing scale of URL queries on a single Docker instance.

Moreover, we estimate the number of safe browsing requests that our PPSB service can handle. First, we calculate the number of requests that the PPSB server can process in a second. The main bottleneck is in the time-consuming generation of the OPRF token. We obtain this number using the JMeter [59], and we denote this number as N_{ps}. Second, our PPSB server only gets involved when there is a match in the prefix filtering, either because the URL is unsafe, or the prefix of the hash value of the URL happens to be in the local prefix database. Remember that, the local prefix database contains the 32-bit prefix of the hash value of unsafe URLs, and there is a chance that the benign URL finds a match in the database. To this end, we use the URLs from *Top 1 Million Sites* [24], together with the randomly selected unsafe URLs from blacklist providers, to measure the number of matches of these URLs in the local prefix database. We deliberately set the percentage of unsafe URLs from 1% to 0.01% to understand the situations in different scenarios. We use N_{total} and N_{match} to denote the number of total URLs and the number of matches in the local prefix database. Third, we calculate the result based on the following equation: $\frac{N_{total} \cdot N_{ps}}{N_{match}}$. Fig. 11 shows the estimated number of safe browsing requests in one second that one PPSB server can support (y-axis) in different percentage of unsafe URLs (x-axis). Note that the real percentage of unsafe URL is

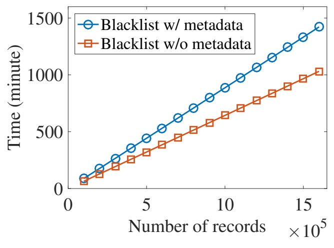


Fig. 13. Preparation time of encrypted blacklist and hash prefixes in one thread.

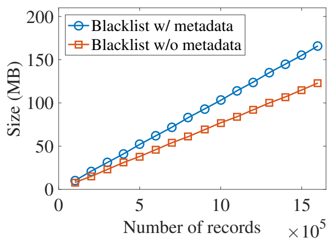


Fig. 14. Size of encrypted blacklist and hash prefixes in uncompressed JSON file.

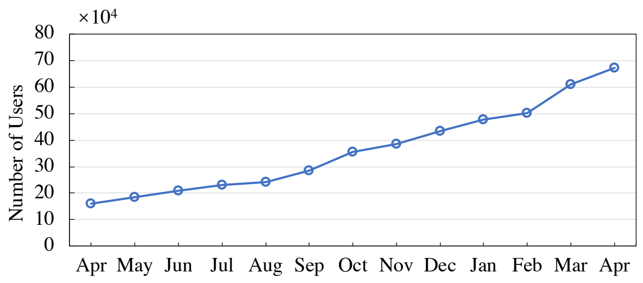


Fig. 15. The growing user base of the Chrome extension "Windows Defender Browser Protection" from Microsoft since its first release in April 2018.

even less than 0.05% reported by Google [40], and our PPSB service can support a large scale requests in this scenario. In addition, Fig. 12 further shows the average latency under the influence of increasing scale of URL queries on a single server instance. We can observe that more requests could be transparently supported by using more threads. Here, we emphasize that the server-side operation is necessary for protecting blacklist providers' proprietary assets (the list of unsafe URLs).

Preparation of the Encrypted Blacklist and Hash Prefixes. We also evaluate the preparation time and the size of the encrypted blacklist and the hash prefixes in our current prototype. Regarding the three real datasets, PPSB servers spend 12 minutes for 14,583 records from PhishTank, 25 minutes for 27,013 records from MalwareDomains, and 1,015 minutes for 1.5 million records in Shallalist when using a single thread, where the prepared data are 1.11 MB, 2.79 MB, and 119.5 MB, respectively. Specifically, Fig. 13 shows the preparation time of the Shallalist in a single thread, which is at the same million level as GSB [40]. We emphasize that this is a one-time cost for each batch of unsafe URLs and can be further accelerated via server-side parallel computing. Fig. 14 shows the size of prepared data (i.e., the encrypted blacklist and its corresponding hash prefixes) in the uncompressed JSON format. We particularly choose JSON format for quick loading and processing in JavaScript. To reduce the bandwidth consumption, we further integrate the data compression technique (i.e., gzip) into PPSB service, so the actual transmitted data are about 0.61 MB, 1.34 MB and 65.5 MB for the three sources. Although this inevitably introduces extra client-side overhead for data decompression, it is indeed fast and unnoticeable, say less than 20 ms for 1.34 MB and 670 ms for 65.5 MB.

In summary, our PPSB prototype on the server side can support a relatively large scale of safe browsing requests, even in the case of two threads. As a future direction, we will continue to find ways to accelerate the speed of building encrypted blacklist and generating OPRF tokens, e.g., refactoring the server-side code in C++ or Go.

8 Related Work

Popular Safe Browsing Services. Our work is closely related to existing SB services adopted by popular web browsers [3], [4], [5], [21], [25]. First, the GSB has a huge impact on several influential browsers. Its current version (v4) consists of the Lookup API and Update API. Particularly, the Lookup API receives the URL in cleartext on the server side, which accentuates the privacy concern in spite of its simplicity. The Update API endeavors to strengthen protection on users' privacy to some extent by infrequently sending 4- to 32-byte hash prefixes of URLs that have passed through the local filter with the server. This Update API is adopted by Chrome, Firefox [9], Safari [10], etc.

Besides that, *Windows Defender SmartScreen* has also been integrated with several Microsoft products such as Edge, IE, and the Windows operating system since its initial release in IE7 [4]. Roughly the same procedures are performed within the extension and those Microsoft products. Although Microsoft stated that sensitive information shared with Microsoft would not be abused to identify, contact or target advertising to the user [60], [61], [62], URLs beyond the scope of a client-side whitelist (that contains a list of safe URLs) are visible to Microsoft.

Moreover, *Fraud and Malware Protection* has been featured in Opera since its early version 9.10 [21]. Prior to Opera 9.50, the domain name of the URL was required to be sent to Opera's server in cleartext, together with a hash of the full URL, which would be checked against some lists of unsafe URLs provided by the third-party blacklist providers GeoTrust and PhishTank [63]. In its subsequent releases, Opera kept this service flow but renewed the blacklist providers from time to time [7], [63], [64], e.g., GeoTrust [65], PhishTank [15], Netcraft [26], and TRUSTe [66]. Note that many recent reports indicate that Opera is now using the GSB, just like Firefox and Safari [12], [67].

In addition, *Yandex Safe Browsing* (YSB) is another known SB service, adopted by the Yandex Browser [25]. YSB follows basically the same procedures as the GSB, but includes its own blacklists other than Google's [12], [68].

Despite the pre-stated privacy policy statement associated with all these widely adopted SB services, potential vulnerability of URL re-identification can be abused by leveraging the shared information [12], [13] as mentioned before.

Searchable Encryption. Our current design leverages the similar idea of searchable encryption (SE) techniques to store the encrypted blacklist and to perform detection on users' client. In principle, most SE schemes surveyed in [69] and many recent ones (e.g., [37], [38], [39]) are applicable to build encrypted yet queryable blacklist as used in our design, explicitly the case with metadata. These SE schemes (just to name a few) focus on improving locality and throughput [19], supporting boolean queries [32], [70] and similarity queries [71], [72], and ensuring forward (and/or backward) security in the dynamic update [37], [38], [39].

To the best of our knowledge, we are the first that apply this technique to the safe browsing scenario. To keep a small memory footprint, we customize the efficient scheme proposed by Cash et al. [19], which is based on a simple yet generic dictionary. Different from its the original scheme, we incorporate an OPRF protocol to enable the client-side extension to perform detection locally, without revealing the private inputs to other parties, i.e., the secret key of the blacklist provider and the queried URLs of the users.

9 Conclusion

PPSB is an open and flexible platform for safe browsing with the guaranteed privacy of users and blacklist providers. In PPSB, the URLs (or even the hash prefixes) to be checked (or vetted) never leave the browser in cleartext. The third-party blacklist providers can contribute their update-to-date lists of unsafe URLs in a private and easy manner. And users can switch different blacklist providers flexibly and obtain updated blacklists automatically. The comprehensive evaluation of our full-fledged and easy-to-use prototype with real datasets demonstrated the efficiency and effectiveness of our design. All the resources, such as code, extension, and Docker image, are available for public use.

Acknowledgments

This work was supported in part by Research Grants Council of Hong Kong under Grant CityU 11212717, and Grant CityU C1008-16G, in part by the National Science Fund for Distinguished Young Scholars (61725205), the National Key R&D Program of China (2018YFB2100800), and the National Natural Science Foundation of China (61572412, 61872438, 61822207, U1636219, and 61772428), in part by the Fundamental Research Funds for the Central Universities under Grant 3102019QD1001, in part by the Equipment Pre-Research Joint Fund of Ministry of Education of China (Youth Talent) under Grant 6141A02033327, and in part by the Outstanding Youth Foundation of Hubei Province under Grant 2017CFA047.

References

[1] L. Lu, V. Yegneswaran, P. Porras, and W. Lee, "Blade: an attack-agnostic approach for preventing drive-by malware infections," in *Proc. of ACM CCS*, 2010.

[2] K. Thomas, F. Li, A. Zand, J. Barrett, J. Ranieri, L. Invernizzi, Y. Markov, O. Comanescu, V. Eranti, A. Moscicki *et al.*, "Data breaches, phishing, or malware? understanding the risks of stolen credentials," in *Proc. of ACM CCS*, 2017.

[3] "Google Safe Browsing," https://safebrowsing.google.com, 2018.

[4] "Evolving Microsoft SmartScreen to protect you from drive-by attacks," https://blogs.windows.com/msedgedev/2015/12/16/smartscreen-drive-by-improvements/#g4WsVdZs8AiAoUsc.97, 2015.

[5] "Safe Browsing Update API (v4)," https://developers.google.com/safe-browsing/v4/update-api, 2018.

[6] T. Bujlow, V. Carela-Español, J. Sole-Pareta, and P. Barlet-Ros, "A survey on web tracking: Mechanisms, implications, and defenses," *Proceedings of the IEEE*, vol. 105, no. 8, pp. 1476–1510, 2017.

[7] Opera, "Opera's fraud and malware protection," https://www.opera.com/help/tutorials/security/fraud/, 2018.

[8] "Safe Browsing Lookup API (v4)," https://developers.google.com/safe-browsing/v4/lookup-api, 2018.

[9] "MozillaWiki: Security/Safe Browsing," https://wiki.mozilla.org/Security/Safe_Browsing, 2018.

[10] C. Hoffman, "How to optimize safari for maximum privacy," https://www.howtogeek.com/103622/how-to-optimize-safari-for-maximum-privacy, 2017.

[11] S. Somogyi and A. Miller, "Safe browsing: Protecting more than 3 billion devices worldwide, automatically," https://www.blog.google/topics/safety-security/safe-browsing-protecting-more-3-billion-devices-worldwide-automatically/, 2017.

[12] T. Gerbet, A. Kumar, and C. Lauradoux, "A privacy analysis of google and yandex safe browsing," in *Proc. of IEEE/IFIP DSN*, 2016.

[13] L. Demir, A. Kumar, M. Cunche, and C. Lauradoux, "The pitfalls of hashing for privacy," *IEEE Communications Surveys Tutorials*, vol. 20, no. 1, pp. 551–565, 2018.

[14] "Google chrome privacy notice," https://www.google.com/intl/en/chrome/privacy/#safe-browsing-policies, 2018.

[15] "Downloadable databases published by phishtank," https://www.phishtank.com/developer_info.php, 2018.

[16] "Malware domain blocklist by riskanalytics," http://www.malwaredomains.com/, 2018.

[17] "BROWSER SECURITY COMPARATIVE ANALYSIS Socially Engineered Malware Blocking," https://www.nsslabs.com/index.cfm/_api/render/file/?method=inline&fileID=A0508430-5056-9046-93AE568F20F794C6, 2018.

[18] J. Burns, D. Moore, K. Ray, R. Speers, and B. Vohaska, "Ec-oprf: Oblivious pseudorandom functions using elliptic curves." *IACR Cryptology ePrint Archive*, 2017.

[19] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation." in *Proc. of NDSS*, 2014.

[20] "Windows Defender Browser Protection," https://browserprotection.microsoft.com/learn.html, 2018.

[21] R. Wagner, "Opera 9.10 released with fraud protection," http://cybernetnews.com/opera-910-released-with-fraud-protection, 2006.

[22] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proc. of ACM ISSTA*, 2007.

[23] "Shalla's blacklists," http://www.shallalist.de/, 2018.

[24] "Top 1 million site scans," https://scans.io/study/scott-top-one-million, 2018.

[25] "Yandex Safe Browsing API," https://tech.yandex.com/safebrowsing, 2018.

[26] "Netcraft," https://www.netcraft.com, 2018.

[27] "Squidblacklist," https://www.squidblacklist.org, 2018.

[28] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[29] R. Nojima and Y. Kadobayashi, "Cryptographically secure bloom-filters." *Trans. on Data Privacy*, vol. 2, no. 2, pp. 131–139, 2009.

[30] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Dupless: Server-aided encryption for deduplicated storage," in *Proc. of USENIX Security*, 2013.

[31] F. Armknecht, J.-M. Bohli, G. O. Karame, and F. Youssef, "Transparent data deduplication in the cloud," in *Proc. of ACM CCS*, 2015.

[32] S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, "Outsourced symmetric private information retrieval," in *Proc. of ACM CCS*, 2013.

[33] X. Yuan, X. Wang, C. Wang, C. Yu, and S. Nutanong, "Privacy-preserving similarity joins over encrypted data," *IEEE Trans. on Information Forensics and Security*, vol. 12, no. 11, pp. 2763–2775, 2017.

[34] H. Cui, Y. Zhou, C. Wang, Q. Li, and K. Ren, "Towards privacy-preserving malware detection systems for android," in *Proc. of IEEE ICPADS*, 2018, pp. 545–552.

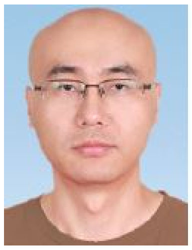
[35] T. Meskanen, J. Liu, S. Ramezanian, and V. Niemi, "Private membership test for bloom filters," in *Proc. of IEEE Trustcom/BigDataSE/ISPA*, vol. 1, 2015, pp. 515–522.

[36] S. Ramezanian, T. Meskanen, M. Naderpour, and V. Niemi, "Private membership test protocol with low communication complexity," in *Proc. of International Conference on Network and System Security (NSS)*, 2017, pp. 31–45.

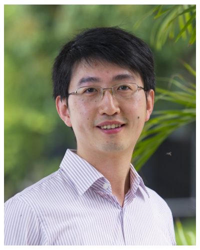
[37] R. Bost, "$\sigma o \varphi o \varsigma$–forward secure searchable encryption," in *Proc. of ACM CCS*, 2016.

[38] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim, "Forward secure dynamic searchable symmetric encryption with efficient updates," in *Proc. of ACM CCS*, 2017.
[39] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and backward private searchable encryption from constrained cryptographic primitives," in *Proc. of ACM CCS*, 2017.
[40] Google, "Google transparency report - safe browsing. (may 2018)," https://transparencyreport.google.com/safe-browsing/overview?hl=en&unsafe=dataset:1, 2018.
[41] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: improved definitions and efficient constructions," in *Proc. of ACM CCS*, 2006.
[42] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proc. of ACM CCS*, 2012.
[43] Y. Zhang, J. Katz, and C. Papamanthou, "All your queries are belong to us: The power of file-injection attacks on searchable encryption," in *Proc. of USENIX Security*, 2016.
[44] "Bloom Filter in JavaScript," https://www.npmjs.com/package/bloomfilter, 2018.
[45] "Create a smaller alternative to the bloom filter for safe-browsing's in-memory structure," https://bugs.chromium.org/p/chromium/issues/detail?id=71832, 2011.
[46] "Set Object in JavaScript," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set, 2018.
[47] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proc. of ACM CoNEXT*, 2014.
[48] T. Berners-Lee, R. Fielding, and L. Masinter, "Rfc 2396: Uniform resource identifiers (uri): Generic syntax, august 1998," *Status: Draft Standard*, 2007.
[49] "Stanford JavaScript Crypto Library," http://bitwiseshiftleft.github.io/sjcl/, 2018.
[50] "Nginx," https://nginx.org/, 2018.
[51] "HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer," http://www.haproxy.org/, 2018.
[52] "Desktop vs mobile market share worldwide," http://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide/, 2019.
[53] Q. Wang, M. Du, X. Chen, Y. Chen, P. Zhou, X. Chen, and X. Huang, "Privacy-preserving collaborative model learning: The case of word vector training," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 12, p. 2381–2393, 2018.
[54] H. Cui, X. Yuan, Y. Zheng, and W. Cong, "Towards encrypted in-network storage services with secure near-duplicate detection," *IEEE Transactions on Services Computing*, vol. PP, no. 99, pp. 1–1, 2018.
[55] S. Hu, L. Y. Zhang, Q. Wang, Z. Qin, and C. Wang, "Towards private and scalable cross-media retrieval," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, pp. 1–1, DOI: 10.1109/TDSC.2019.2 926 968, 2019.
[56] H. Cui, H. Duan, Z. Qin, C. Wang, and Y. Zhou, "Speed: Accelerating enclave applications via secure deduplication," in *Proc. of IEEE ICDCS*, 2019.
[57] "Average page load times for 2018 – how does yours compare?" https://www.machmetrics.com/speed-blog/average-page-load-times-websites-2018/, 2018.
[58] "The top 500 sites on the web," https://www.alexa.com/topsites, 2018.
[59] "Apache jmeter," https://jmeter.apache.org, 2018.
[60] "Internet Explorer 11 privacy statement," https://privacy.microsoft.com/nl-NL/ie11-win8-privacy-statement, 2014.
[61] "Microsoft Edge, browsing data, and privacy," https://privacy.microsoft.com/en-us/windows-10-microsoft-edge-and-privacy, 2018.
[62] "Microsoft Privacy Statement," https://privacy.microsoft.com/en-us/privacystatement, 2018.
[63] Opera, "Opera fraud protection prior to 9.5," https://www.opera.com/help/tutorials/security/fraud/910, 2018.
[64] ——, "Opera fraud protection," https://www.opera.com/help/tutorials/security/fraud/950/, 2018.
[65] "Geotrust," https://www.geotrust.com, 2018.
[66] "Trustarc: the new truste," https://www.trustarc.com, 2018.
[67] D. Aleksandersen, "Most of the alternate web browsers don't have fraud and malware protection," https://www.ctrl.blog/entry/fraud-protection-alternate-browsers, 2017.
[68] RBLTracker, "Blacklisted websites - google and yandex safe browsing databases," https://rbltracker.com/blog/2016/08/blacklisted-websites-google-and-yandex-safe-browsing-databases, 2016.
[69] C. Bösch, P. Hartel, W. Jonker, and A. Peter, "A survey of provably secure searchable encryption," *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, p. 18, 2015.
[70] S.-F. Sun, J. K. Liu, A. Sakzad, R. Steinfeld, and T. H. Yuen, "An efficient non-interactive multi-client searchable encryption with support for boolean queries," in *Proc. of ESORICS*, 2016.
[71] M. Kuzu, M. S. Islam, and M. Kantarcioglu, "Efficient similarity search over encrypted data," in *Proc. of IEEE ICDE*, 2012.
[72] Q. Wang, M. He, M. Du, S. S. M. Chow, R. W. F. Lai, and Q. Zou, "Searchable encryption over feature-rich data," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 3, p. 496–510, 2018.

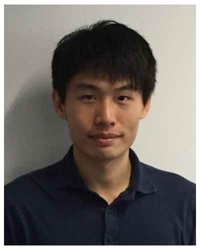
Helei Cui received the B.E. degree in software engineering from Northwestern Polytechnical University in 2010, the M.S. degree in information engineering from The Chinese University of Hong Kong in 2013, and the Ph.D. degree in computer science from City University of Hong Kong in 2018. He is currently an Associate Professor with the School of Computer Science, Northwestern Polytechnical University, China. His research interests include cloud security, mobile security, and multimedia security.



Yajin Zhou received the Ph.D. degree in computer science from North Carolina State University, Raleigh, NC, USA. He is currently a ZJU 100 Young Professor with the School of Cyber Science and Technology, and the College of Computer Science and Technology, Zhejiang University, China. His research mainly focuses on smartphone and system security, such as identifying real-world threats and building practical solutions, mainly in the context of embedded systems (or IoT devices).



Cong Wang (SM'17) is currently an Associate Professor in the Department of Computer Science, City University of Hong Kong. He received his Ph.D. degree in the Electrical and Computer Engineering from Illinois Institute of Technology, USA, M.E. degree in Communication and Information System, and B.E. in Electronic Information Engineering, both from Wuhan University, China. His current research interests include data and computation outsourcing security in the context of cloud computing, blockchain and decentralized application, network security in emerging Internet architecture, multimedia security, and privacy-enhancing technologies in the context of big data and IoT. He is one of the Founding Members of the Young Academy of Sciences of Hong Kong. He received the Outstanding Research Award in 2019, the Outstanding Supervisor Award in 2017, and the President's Awards in 2016 from City University of Hong Kong. He is a co-recipient of the Best Student Paper Award of IEEE ICDCS 2017, the Best Paper Award of IEEE ICPADS 2018, MSN 2015 and CHINACOM 2009. His research has been supported by multiple government research fund agencies, including National Natural Science Foundation of China, Hong Kong Research Grants Council, and Hong Kong Innovation and Technology Commission. He serves/has served as associate editor for IEEE Transactions on Dependable and Secure Computing (TDSC), IEEE Internet of Things Journal (IoT-J) and IEEE Networking Letters, and TPC co-chairs for a number of IEEE conferences/workshops. He is a senior member of the IEEE, and member of the ACM.



Xinyu Wang received the B.E. degree in software engineering from East China Jiaotong University, in 2011. He is currently working towards the Ph.D. degree in the Department of Computer Science, City University of Hong Kong. He worked for Tencent as a software engineer from 2011 to 2013. His research interests include cloud computing, network security, and big data.

Yuefeng Du is currently working towards the Ph.D. degree in the Department of Computer Science, City University of Hong Kong, where he obtained his B.S. degree in computer science in 2018. His research interests include cloud storage security, computer security, and artificial intelligence security.

Qian Wang (SM'18) is currently a Professor with the School of Cyber Science and Engineering, Wuhan University. He received the B.S. degree from Wuhan University, China, in 2003, the M.S. degree from the Shanghai Institute of Microsystem and Information Technology, Chinese Academy of Sciences, China, in 2006, and the Ph.D. degree from the Illinois Institute of Technology, USA, in 2012, all in electrical engineering. He received National Science Fund for Excellent Young Scholars of China in 2018. He is also an expert under the National "1000 Young Talents Program" of China. His research interests include AI security, data storage, search and computation outsourcing security and privacy, wireless systems security, big data security and privacy, and applied cryptography. He was a recipient of the IEEE Asia-Pacific Outstanding Young Researcher Award 2016. He was also a co-recipient of several Best Paper and Best Student Paper Awards from IEEE ICDCS'17, IEEE Trust-Com'16, WAIM'14, and IEEE ICNP'11. He serves as an Associate Editor for the IEEE Transactions on Dependable and Secure Computing (TDSC) and the IEEE Transactions on Information Forensics and Security (TIFS). He is a senior member of the IEEE, and member of the ACM.