Heterogeneous-PAKE: Bridging the Gap between PAKE Protocols and Their Real-World Deployment

Rong Wei*
State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
weirong1130@iie.ac.cn

Fangyu Zheng†
State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
zhengfangyu@iie.ac.cn

Lili Gao*
State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
gaolili1994@iie.ac.cn

Jiankuo Dong
School of Computer Science, Nanjing University of Posts and Telecommunications
djiankuo@njupt.edu.cn

Guang Fan*
State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
fanguang@iie.ac.cn

Lipeng Wan*
State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
wanlipeng@iie.ac.cn

Jingqiang Lin
School of Cyber Security, University of Science and Technology of China
linjq@ustc.edu.cn

Yuewu Wang*
State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
wangyuewu@iie.ac.cn

ABSTRACT

Two entities, who only share a password and communicate over an insecure channel, authenticate each other and agree on a large session key for protecting their subsequent communication. This is called the password-authenticated key exchange (PAKE) protocol. PAKE protocol has been considered a suitable substitute for the prevailing hash-based authentication which is vulnerable to various attacks. However, vendors are discouraged by both its prohibitively computational overheads as well as integrating costs, leading to its limited use since being proposed.

After carefully analyzing the general workflow of PAKE protocols, we present Heterogeneous-PAKE, an entire PAKE stack with high-performance and compatibility for both client-side and server-side for Web systems. Using SRP and SPAKE2+ as case studies, we conduct a series of comprehensive experiments, especially comparing with the conventional hash-based solutions to evaluate the Heterogeneous-PAKE. The implementation harvests high throughput on the server-side with over 240k, 70k, 30k, and 1,650k operations per second for SRP-1024, SRP-1536, SRP-2048, and SPAKE2+ respectively. Meanwhile, on most testing platforms, the latency is well controlled within user-acceptable bounds, especially the SPAKE2+ whose delay is less than 3x that of a traditional authentication approach based on Bcrypt. The empirical results demonstrate that the Heterogeneous-PAKE is a very economical (with only a GPU-ready server) and convenient (with an easy-to-integrate software stack without user participation or database rebuilding) solution for upgrading existing systems with high-performance PAKE services.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Applied computing** → **Service-oriented architectures**.

KEYWORDS

PAKE protocol, password authenticating, heterogeneous computing model, SRP, SPAKE2+

ACM Reference Format:
Rong Wei, Fangyu Zheng, Lili Gao, Jiankuo Dong, Guang Fan, Lipeng Wan, Jingqiang Lin, and Yuewu Wang. 2021. Heterogeneous-PAKE: Bridging the Gap between PAKE Protocols and Their Real-World Deployment. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3485832.3485877

1 INTRODUCTION

Passwords constitute the most ubiquitous means of authentication on the Internet, from the common to the most sensitive applications. A general password-based authentication practice relies on SSL/TLS [37] and the slow hash functions (e.g., Bcrypt [35] and Scrypt [33]): the user sends her (his) password to the server under over an SSL/TLS-protected channel, and then the server decrypts the password and verifies against a one-way image typically computed via slow hash functions. The password image of each user is stored on the server. There are two obvious disadvantages of this approach:

- Although storing the password images on the server-side, it is still vulnerable to dictionary attacks once the database leaks. [43] summarized 56 biggest data breaches, in

* The authors are also with School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
† Fangyu Zheng is the corresponding author.

ACSAC '21, December 6–10, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3485877

which Twitter, Facebook and Yahoo, as famous international IT vendors in the world, were reported to be victims of data breaches, leading to compromises of 300 million to 3 billion users' private data, including plaintext passwords or hashed passwords. Worse still, since password reuse is rampant, any data breach would make those innocent, heavily-guarded sites potentially vulnerable. With credential stuffing techniques, attackers could easily expand damages to websites not compromised.

- The password (or its hashed image) in travel is exposed to eavesdroppers and man-in-the-middle (MitM) attacks [5]. Although most websites are protected by SSL/TLS connections, security breaks if the SSL/TLS channel is established with a compromised server's private key (a widespread concern given today's too-common PKI failures [39]). Even if the TLS channel was not corrupted, passwords would still be decrypted after passing through the TLS offload gateway, and then travel in plaintext through the internal network.

1.1 PAKEs and the Challenges When Applied

Considering these problems, researchers proposed password authenticated key exchange (PAKE) protocols [4], which are geared to prevent the above-mentioned attacks. As the name suggests, a PAKE protocol is intended to establish a strong session key for participants based on a (weak) password.

According to the knowledge of the password hold by each participant, a PAKE protocol can be labeled as symmetric (or balanced) PAKE (bPAKE) or asymmetric (or augmented) PAKE (aPAKE). While the password is shared by all of the participants, bPAKE protocols do not apply to the client-server setting. In contrast, an aPAKE protocol not only prevents passwords from eavesdropping on channels but from data leakages on the server-side. In this paper, we concentrate on aPAKE schemes, which are more suitable for Web applications. Compared with traditional password-based login, instead of authenticating the user by validating her (his) password with the hash stored on the server-side, an aPAKE protocol is characterized with the following two properties:

(1) Messages are randomized and thus, immune to insecure communication channels like the Internet.
(2) The server side does not keep a password or its hash, instead, it holds a much more complicated mapping (in other words, a verifier) of the password. The security strength of a verifier is based on mathematical hard problems, thus the passwords are protected against dictionary attacks [29].

Over the past three decades, PAKEs have been heavily studied but only been used in quite limited ranges compared with the huge number of websites. As far as we know, Apple employed SRP [47] for protecting iCloud security codes and recently used SPAKE2+ [42] for car keys security [15]. 1Password used SRP along with TLS/SSL to secure authentication between client and server without storing plaintext user passwords [12]. Meanwhile, other websites are barely reported for PAKE usage, Facebook [1] and Google [2] are even reported to store passwords in plaintext. The reasons for PAKE's limited real-world usage are manifold, among which performance issues and migration cost for the existent systems are the main concerns. Technical challenges include:

- Constructed over public-key cryptography, a PAKE protocol is characterized by far higher workloads than hash-based authentications.
- Besides significant performance penalties, vendors have to pay heavy taxes to upgrade their systems for PAKE service. In contrast to simple cryptographic primitives, e.g., digital signature and data encryption, a PAKE protocol consists of multiple roundtrips requiring costly modifications to the server-side.
- Compared with hash-based manners, an aPAKE protocol requires rich clients to perform complex computations, whereas diverse client-side environments bring more compatibility issues.

1.2 Contributions

In this paper, after carefully analyzing the general workflow of aPAKE protocols, we present Heterogeneous-PAKE, an entire PAKE stack for Web systems that offers high-performance and compatibility for both client-side and server-side, including cryptographic primitives, protocol implementations and network scheduling.

The main goal of this work is to bridge the gap between PAKE protocols and their real-world deployment at the lowest possible cost. More specifically, our contributions are four-fold:

(1) Firstly, we put forward an easy-to-integrate PAKE service framework for Web applications, minimizing the vendors' costs for system upgrading.
(2) Secondly, on the server side, we provide a full PAKE implementation in SIMD and SIMT computing fashions to process highly concurrent user requests. The server-side also integrates heterogeneous computing to the Web systems with minimal modifications by a finely scheduled network processing architecture. While the framework harvests low latency as well as high throughput, the extraordinary performance mainly comes from COTS products, including a desktop-class CPU and an NVIDIA graphics card.
(3) Thirdly, on the client side, we implement a JavaScript-based library including the fundamental cryptographic primitives for both DL and EC-based PAKE, as well as the high-level PAKE protocols.
(4) Finally, targeting two representative aPAKE protocols, SRP and SPAKE2+, we build a high-performance aPAKE server and corresponding components for Web applications. The implementation harvests high throughput on the server-side with over 240k, 70k, 30k and 1,650k operations per second for SRP-1024, SRP-1536, SRP-2048 and SPAKE2+ respectively. In the meanwhile, the client-side's latency is well controlled within user-acceptable bounds, especially the SPAKE2+ implementation whose delay is less than 3x that of a traditional authentication approach based on Bcrypt and does not exceed 0.4s on most platforms even facing the concurrency of over 1600k requests per second.

1.3 Outline

The next section begins with the background and requisite knowledge. Section 3 illustrates the overall framework of the prototype system. Section 4 details the design and optimizations we adopted on different aspects. In Section 5 we conduct a series of experiments and performance evaluations. Finally, we draw conclusions in Section 6.

2 BACKGROUND

This section introduces basic knowledge of PAKE protocols, demonstrates two target protocols, and illustrates features of involved parallel platforms.

2.1 PAKE Protocol

A PAKE scheme is an interactive protocol that allows its participants to authenticate each other and agree on a shared cryptographic key with a (weak) password.

Augmented PAKE (aPAKE) protocols [17, 40, 42, 46] are further designed to keep an individual user's password secure against server compromise in client-server scenarios which are quite common in the real world.

Although PAKE protocol achieves strong security with weak passwords, it has been hampered from widespread use by performance issues. Besides expensive mathematical computations and multiple interactions, some schemes introduce additional complexity to circumvent existing patents. Table 1 lists the main overheads of representative aPAKE protocols, including the number of modular exponentiation for discrete logarithm (DL) setting or scalar multiplications for elliptic curve (EC) setting. Since the concrete operations of a PAKE scheme vary in different publications, our data comes from IEEE P1363.2 [13] and Internet drafts [42]. Particular attention is paid to fixed-base operations which tend to be accelerated with pre-computations. In order to solve this dilemma, we design an entire PAKE stack for Web systems and implement SRP-3 [47] and SPAKE2+ [42] as two case studies.

Table 1: Overhead of aPAKE Protocols

Protocol	Setting	Client		Server	
		Fixed-base	Total	Fixed-base	Total
AMP[21]	DL/EC	2	4	0	3
BSPEKE2[16]	DL/EC	2	6	1	5
PAKZ[23]	DL/EC	1	2	1	2/1
SPAKE2+	DL/EC	1	8/6	1	6/5
SRP-3	DL	2	3	1	3
SRP-5[45]	EC	2	4	1	3
SRP-6[46]	DL	2	3	1	3

Secure Remote Password (SRP) protocol, a verifier-based PAKE scheme proposed by T Wu [47], is suitable for negotiating secure connections using a user-supplied password while eliminating the security problems traditionally associated with reusable passwords. Figure 1 presents steps of SRP-3 which take three round trips to finish a key exchange and effectively conveys a zero-knowledge password proof from the user to the server. As the most representative one of industrialized aPAKE protocols, SRP is used for account validation in keychain recovery and signing into shared devices in iPhone [15]. Besides, SRP is integrated into TLS [46].

SPAKE2+ is the augmented version of SPAKE2 [3] protocol which is compatible with any prime order group and is proven secure in the Universal Composable (UC) model [6]. While SRP works on a ring and is hence uneasy for transplanting to an EC group, SPAKE2+ is EC-compatible and does not require any full domain hash function or ideal ciphers onto EC groups. When implemented with EC, the protocol runs as Figure 2 shows. The verifier of SPAKE2+ consists of an element w_0 on $GF(p)$ and an

	Client		Server
1.	Input (ID, π)	$\overset{ID}{\rightarrow}$	(lookup s, v)
2.	$x = H(s, \pi)$	$\overset{s}{\leftarrow}$	
3.	$A = g^a$	$\overset{A}{\rightarrow}$	
4.		$\overset{B,u}{\leftarrow}$	$B = v + g^b$
5.	$S = (B - g^x)^{a+ux}$		$S = (Av^u)^b$
6.	$K = H(S)$		$K = H(S)$
7.	$M_1 = H(A, B, K)$	$\overset{M_1}{\rightarrow}$	(verify M_1)
8.	(verify M_2)	$\overset{M_2}{\leftarrow}$	$M_2 = H(A, B, K)$

H : hash function s : the user's salt stored on server side
π : the user's password g : a generator of GF(n)
x : private key derived from s and π
ID : the user's identity v : $verifier$ derived from g and x,computed as $v = g^x$, stored on server side
a, b, u and v are one-time random numbers

Figure 1: the Workflow of SRP3

EC point L computed as $L = [w_1]P$, where w_1 together with w_0 are derived from the password π, and P is a generator of the EC group G.

	Client		Server
1.	Input (ID, π)	$\overset{ID}{\rightarrow}$	(lookup w_0, L)
2.	$(w_{0s}, w_{1s}) = PBKDF(\pi)$	$\overset{*}{\leftarrow}$	
3.	$w_0 = w_{0s} \bmod p$		
4.	$w_1 = w_{1s} \bmod p$		
5.	$X = [x]P + [w_0]M$	$\overset{X}{\rightarrow}$	$Y = [y]P + [w_0]N$
6.	$Z = [hx](Y - [w_0]N)$	$\overset{Y}{\leftarrow}$	$Z = [hy](X - [w_0]M)$
7.	$V = [hw_1](Y - [w_0]N)$		$V = [hy]L$
8.	$K_a \| K_e = H(TT)$		$K_a \| K_e = H(TT)$
9.	$K_{cA} \| K_{cB} = KDF(Ka)$	$\overset{K_{cA}}{\rightarrow}$	(verify K_{cA})
10.	(verify K_{cB})	$\overset{K_{cB}}{\leftarrow}$	$K_{cA} \| K_{cB} = KDF(Ka)$

π : the user's password L : the *verifier* of π, computed as $L = [w_1]P$
H : hash function ID : the user's identity, such as username and Email
P : a generator of G h : the cofactor of G
M and N are fixed elements of G x and y are one-time random elements.
TT : concatenation of ID, $Server$, M, N, X, Y, Z, V, w_0

Figure 2: the Workflow of SPAKE2+

2.2 Parallel Platforms

Facing up with the growing demand for greater computing powers across industry segments, vendors tend to equip their processors with vector instruction sets, such as AMD 3DNow [30], ARM NEON [36], etc. Intel also announced its MMX/SSE/AVX [18, 32] series. A vector instruction performs the same operation to multiple elements resided on a vector register in parallel, which is also known as single instruction, multiple data (SIMD). As the latest, also the most powerful version of the AVX series up to now, AVX-512 [7] provides 8-way 64-bit (or 16-way 32-bit) instructions which are of great importance for multi-precision arithmetic in the public-key cryptosystem.

While vector instruction brings several-fold speedup to CPU programs, this is far from dealing with highly concurrent tasks. Consequently, researchers attempt to exploit the powerful computing ability of graphic processing units (GPUs) to handle massive requests effectively.

GPUs consist of a large number of cores, processing parallel tasks by broadcasting an instruction to every executing unit, which is called single instruction, multiple threads (SIMT). When used for computations other than graphics rendering, GPUs act as a co-processor with built-in memory and processing units. NVIDIA released Compute Unified Device Architecture (CUDA) [38], a programming framework that enables developers to dramatically speed up computing applications by harnessing the power of GPUs. Some excellent works for GPU-accelerated computation have been delivered in recent years [8–11, 31].

3 OVERVIEW

As we aim to provide efficient aPAKE services for Web applications, this section illustrates our solution and considerations.

3.1 Design Goal

Since our goal is proposing a solution for integrating aPAKE services to existing Web applications (which may be a large-scale website with massive user data), instead of building a new website with aPAKE functionality, some trivial issues should be taken into account. The main requirements of our scheme include:

- **easy-to-integrate.** The vendors could upgrade their applications with minimal costs.
- **extensible.** It should be convenient to extend the system to clustered Web servers, and it should not produce an extra price when the cluster scale grows.
- **high-performance.** The system shall adapt to dynamic workloads. Concretely, it achieves high throughput when facing high concurrency requests and low latency for sporadic tasks.

3.2 Workflow Analysis

PAKE schemes are characterized by multiple interactions between the participants in a single execution. Despite the complexity, the standard authentication of an industrialized aPAKE protocol could be summarized as Figure 3, which consists of three general stages:

(1) **preamble exchange.** In this stage, the user types in her (or his) ID and password in a form. Then the ID and negotiable information (Inf_C) like cipher suite and protocol version are sent to the server-side. Upon reception of the ID, the server retrieves its verifier (and salt if necessary) from the database accordingly and returns requisite information (Inf_S) to the client-side.
(2) **protocol setup.** By exchanging key derivation materials (KD_C and KD_S), participants compute their shared secret and then derive session keys (K_C or K_S). The process contains at least one data exchange. K_C and K_S will be equal if the protocol succeeds. For the server-side, all mathematical computations occur in this stage.
(3) **key confirmation (optional).** Participants, especially servers, are required to confirm the clients' knowledge of the session key before putting it into use. The verification tends to be finished with a hash-based scheme.

A stage does not always mean a roundtrip or an interaction, since there may be multiple roundtrips at a single stage (especially the second phase) of a few protocols. Even for the same PAKE protocol, its communication overhead varies in different publications, e.g. academic papers, standard documents. In this work, we implement protocols according to their definitions in IEEE P1363.2 or Internet drafts.

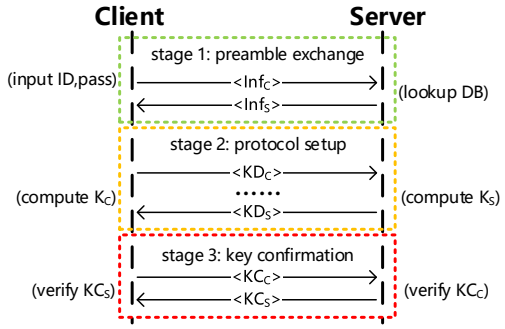


Figure 3: Protocol Flow of aPAKE, where K means "Key", KD means "Key Derivation", and KC means "Key Confirmation"

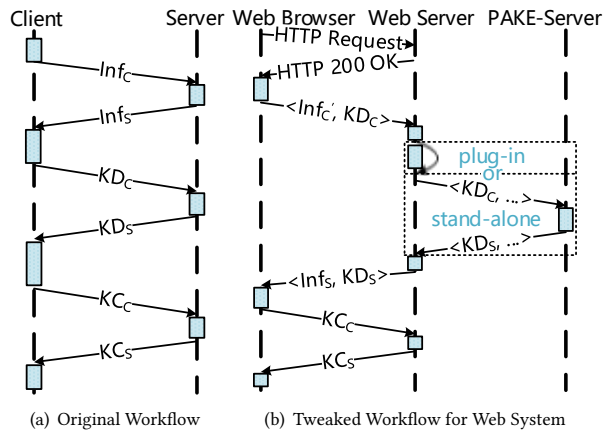
When integrated into Web applications, there is a possibility for further simplifying an aPAKE protocol to two stages without a security breach. Since a Web browser downloads JavaScript programs which hard code co-parameters from the server over HTTP(S) on opening URL, parameter negotiation is no longer a prerequisite. The rest of the preamble exchange could be postponed to the next stage **if KD_C in stage 2 is independent to Inf_S in stage 1**.

Both the two protocols we select for implementation are of three rounds and satisfy the above assumption. Figure 4 shows the original workflow of a 3-round aPAKE protocol and the modified version we proposed for Web systems. The preamble exchange is replaced with the requisite preload of JavaScript and a manual HTTP request. Mathematical computations of server-side aPAKE centralize within the dotted border, which could be finished with a plug-in or an independent server.

A plug-in runs on the target device as a software add-on and has a great advantage of responding speed over remote services for its relatively short invocation path. However, a complicated plug-in brings a noticeable performance penalty with too much occupation of native resources, especially when facing massive tasks. When it comes to integrating a computational expensive service (e.g. SSL management) into existent systems, an independent server is preferred for lower upgrade costs and better performance in high concurrent situations. While aPAKE protocols are computationally intensive and thus naturally suitable for remote serving mode, we provide aPAKE service remotely with a stand-alone server (referred to as PAKE-Server later in this paper). Another consideration is that most COTS servers are headless devices that have no space for a powerful discrete graphics card required by our scheme.

More specifically, when a user opens the website, the browser downloads static resources including the JavaScript program with aPAKE functions. Then the user inputs her (or his) username ID and password P in a form and clicks the "Submit" button . Subsequently, the authentication is carried out as follows:

(1) The Web browser computes KD_C and necessary intermediate variables with ID and P and sends KD_C together with ID to the Web server.
(2) The Web server looks up the user's verifier v (and salt s, if any) from the database. If the retrieval succeeds, it establishes a session for the user and sends v and requisite data in KD_C to PAKE-Server.



(a) Original Workflow (b) Tweaked Workflow for Web System

Figure 4: Workflow of 3-round aPAKE Protocols

(3) PAKE-Server computes KD_S as well as other requisite materials for key derivation and returns them to the Web server.
(4) The Web server derives session key K_S based on data received, and then replies to the browser with KD_S.
(5) The Web browser derives session key K_C, then computes and sends key confirmation material KC_C to the Web server.
(6) The Web server computes key confirmation material KC_S, verifies the equivalence of K_C and K_S with KC_C and KC_S, and then sends KC_S back to the browser. If the key confirmation fails, it destroys the session and clears all variables cached before.
(7) The Web browser confirms the session key based on KC_C and KC_S. If $K_C = K_S$, then output K_C, otherwise the authentication fails.

4 PAKE IMPLEMENTATION

Taking SRP and SPAKE2+ as case studies, this section details the implementations of our scheme in a bottom-up manner. We firstly optimize fundamental primitives for CPU, GPUs as well as Web browsers. Then a heterogeneous protocol stack is constructed over the primitives. After that, we illustrate the up-level design of the PAKE-Server, including I/O processing and task dispatching. Finally, we present the overall framework and give guidelines for applying the scheme to existing Web systems.

4.1 Cryptographic Primitive Optimizations

We provide implementations of each protocol in JavaScript for Web browser, C (with AVX-512 instructions) and CUDA C for PAKE-Server respectively. The task is decomposed into three levels as shown in Figure 5 from a developer's perspective. More specifically, SRP is constructed over $GF(p)$ where p tends to be a generalized prime, and thus Montgomery multiplication [27] is required for modular multiplication and exponentiation. By comparison, we implement SPAKE2+ over Edwards25519 [14] group, which could further be decomposed into $GF(p25519)$ operations where $p25519 = 2^{255} - 19$. While $p25519$ is a pseudo-Mersenne prime, the modular operation could be accelerated with fast reduction algorithm.

4.1.1 JavaScript Implementation. The client-side program is reconstructed over the open-source JavaScript library *sjcl* firstly

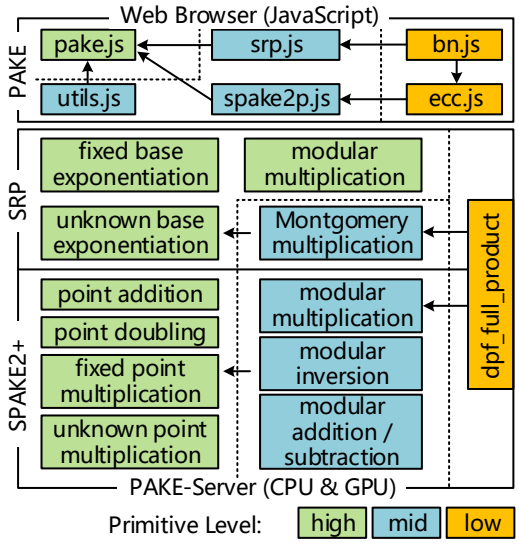


Figure 5: Key Primitives

released by Stark et. al. [41] in 2009. Considering the browser backward compatibility, the client-side protocols are performed by pure JavaScript-based implementation. Despite the excellent compatibility, JavaScript is naturally much slower than native languages and needs to download on each visit to the website. Therefore, the code size should be rigorously restricted to reduce transmission delay. Worse still, affected by the single-thread mode of JavaScript, any synchronized executing would block the page rendering, and hence, we do not suggest time-consuming pre-computations. While the Web Worker API [34] seems to be able to offer acceleration in a background thread, its structured clone brings non-negligible delay to cryptography computations.

Consequently, we decompose *pake.js* into several modules as shown in Figure 5. The basic level consists of two modules, *bn.js* implements finite field operations, and *ecc.js* provides a framework for ECC functionalities. Both the two are built-in modules of *sjcl.js*, and *bn.js* is also a building block for *ecc.js*. While *ecc.js* only provides Weiertrass curves, we additionally implement Edwards25519 curve with extended coordinate system based on the framework.

The instant downloading mode prohibits offline pre-computing skills and large hard-coded lookup-tables. As a result, we do not distinguish fixed base (point) operations from unknown ones and accelerate them with fixed-window [44] uniformly by pre-computing on the fly. Taking EC as an example, a point is defined as an object with a member *multiples* for storing the pre-computed values. *multiples* will be filled during the first point multiplication performed on the object and then be cached for reuse.

Based on the above fundamental modules and other cryptographic primitives (e.g., SHA256 [20], PBKDF2 [28]) provided by *sjcl*, we implement client-side protocols in *srp.js* and *spake2p.js*, and then form into the library *pake.js* with necessary tools defined in *utils.js*. Finally, the modules are compressed into a 61-KB *js* file.

4.1.2 CUDA Implementation. An appropriate sampling routine for arbitrary-precision integers can significantly improve the overall performance of PAKE implementations. Although a large sampling size increases the degree of parallelism, the principle

is under the restriction of the word size of the platform-specific fixed-precision number. To achieve the product of two words without losing precision, the word size tends to be less than half of a machine word. In this work, we adopt a state-of-the-art data representation proposed by N. Emmart et. al. [10] for $GF(p)$ operations on GPUs.

Emmart et. al.'s method: The method samples a large integer in 52-bit limbs, with each limb stored as the mantissa of a double-precision floating-point (DPF) number which is denoted as *double* in most programming languages. Compared with other variable types, the floating-point format and its arithmetic are standardized in IEEE 754 [19], thus behavior diversities of DPF are eliminated on all standard-compliant platforms. With the help of fused-multiply-and-add (FMA) instruction, such sampling strategy conducts a full product of two 52-bit limbs efficiently as shown in Figure 6. The involvement of c1 and c2 could align the valid bits of hi and lo, thus avoid possible overflows in later accumulation. With Montgomery reduction implementation based on the above idea and some additional tricks, the scheme outperforms all of the alternatives proposed for modular exponentiation on GPUs before.

```
1 dpf_full_product(double a_sample, double b_sample)
2 {
3       double hi, lo, c1 = 2^104, c2 = 2^104 + 2^52, sub;
4       hi = __fma_rz(a_samples, b_samples, c1);
5       sub = c2 - hi;
6       lo = __fma_rz(a_samples, b_samples, sub);
7       return (hi, lo);
8 }
```

Figure 6: dpf_full_product for GPUs

For the underlying Edwards25519 curve of SPAKE2+ protocol, we use Gao et. al.'s method (called *DPF-ECC*) [11] which applies Emmart et. al.'s scheme for fast reduction over $GF(p25519)$ with ingenious modifications.

DPF-ECC: The method tweaks Emmart et. al.'s scheme to a mixed-radix representation, sampling a 256-bit number into four 51-bit limbs and a 52-bit limb (also the most significant limb). To keep the alignment of the product of two limbs with unfixed word size, variables c1 and c2 in function dpf_full_product of Figure 13 are set to 2^{103} and $2^{103} + 2^{52}$. Since the modulus, $p25519$ is a pseudo-Mersenne prime which allows fast reduction, outputs of underlying addition and subtraction over $GF(p25519)$ converge to the range $[0, 2^{255} + 4845)$ with only one round of reduction.

As registers are precious resources in GPUs, we split an operand into multiple pieces according to the modulus length of $GF(p)$ and perform operations with the corresponding number of threads in parallel. In contrast, an element of $GF(p25519)$ consists of only five limbs, and thus no longer needs threads co-operation.

4.1.3 AVX-512 Implementation. AVX-512 provides FMA instructions including $vfmadd132pd$, $vfmadd213pd$ and $vfmadd231pd$, making it convenient to transplant Emmart et. al.'s algorithm to CPU. While AVX-512 broadcasts the same instruction to each element of the vector, it multiplies a 52-bit limb b by a with eight 52-bit limbs. As a result, we implement vectorized full product as shown in Figure 13.

Actually, AVX-512 offers an IFMA feature that performs FMA on 52-bit integers. With this new feature, the function in Figure 13 could be implemented simply and elegantly, as shown in Figure 14. However, IFMA has not been supported by any desktop processor yet.

When it comes to $GF(p25519)$, the best sampling strategy seems to be 32-bit or 64-bit limbs, and therefore, a ZMM register could neatly hold two Edwards25519 coordinates. In addition, underlying finite field operations of extended coordinates tend to appear in pairs and are independent of each other. However, some requisite instructions, e.g., *vpadcd*, *vpsbbd*, *vpmulld*, *vpmulhud*, are only available on Intel phi processors which are somehow obsolete comparing to GPUs for the sake of low frequency, high TDP, limited resources and rigorous requirements for hardware platforms. Based on the above fact, we still apply *DPF-ECC* scheme for AVX implementation.

However, *DPF-ECC* cannot be smoothly transplanted to vector instructions, since the mixed-radix representation interrupts the unity of operations on the elements of a vector variable. We preset some important vectors in Table 10 for later computations, with the i-th element of a vectorized variable (i.e. __m512i, __m512d) a denoted as $a[i]$ (just like an array) for convenience.

Pre-computation: The square-and-multiply approach for exponentiation (i.e. b^e) takes l times of squaring and multiplications where l is the bit length of the group modulus. We exploit w-bit fixed-window to accelerate unknown base modular exponentiation, taking l times of squaring, l/w times of memory-access-and-multiply, and extra $2^w - 2$ multiplications for pre-computation. When it comes to fixed bases, a much larger table could be computed offline. With j-th power of $b^{i\times w}$ stored as $pretb[i][j]$ where $0 <= i < \lceil l/w \rceil$ and $0 <= j < 2^w$. After that b^e is calculated as $\prod_{i=0}^{l/w-1} pretb[i][e_{i\times w:(i+1)\times w-1}]$. We store the elements of $pretb$ as *double* type for convenience and the offline file's size $M = 2^w \times \lceil l/52 \rceil \times \lceil l/w \rceil \times 64$ bits.

To accelerate known-point multiplication on EC groups, we pre-compute offline table as depicted in [22] with window size 16. In addition, we compress the coordinates as defined in RFC7748 to save the storage. As a result, the offline file's size is $\lceil 255/16 \rceil \times 32 \times 2^{16}B = 32MB$. As for unknown point multiplication, we use the 4-bit fixed window method for side-channel considerations.

Carry Resolve and Reductions: It is noticeable that *DPF-ECC* brings additional overheads for reductions when implemented with vector instructions for the disunity of its word size. Standard carry resolution approaches are no longer applicable to AVX-based $GF(p25519)$ elements and thus, we exploit *carry-predicting* technique [9] for finite field operations.

Figure 15 gives the vectorized modular addition with *carry-predicting* as example. Note that the last carry resolve from Step 17 to Step 21 is slightly different from the standard *carry-predicting* at the beginning (Line 6 to Line 9). Since the fast reduction in step 15 adds a little number (less than 39) to $c[0]$, bringing no carry immediately to $c[1 \sim 4]$. As a result, we could obtain the information about $generate$ as well as $propagate$ at once by comparing c with the special mask $[2^{51}, 2^{51} - 1, 2^{51} - 1, 2^{51} - 1, 2^{52} - 1, 0, 0, 0]$. Specifically, the least significant bit of $generate$ indicates the only possible carry bit and the left ones imply $propagation$ of $a[1 \sim 4]$.

In traditional ECC-based applications such as digital signature, partially reduced coordinates won't affect the correctness. However, this view does not hold for PAKE protocols in which a coordinate may be input to hash functions. In other words, two secrets congruent modulo q, i.e. $s_1 = s_2 \ mod \ q$ while $s_1 \neq s_2$, would lead to distinct session keys.

There are two intermediate result forms not thoroughly reduced in $GF(p25519)$ operations under the lazy-reduction strategy:

- *Case 1*: $2^{255} <= a < 2^{255} + 4845$, more specifically, $a[4] = 2^{51}$, $a[1] = a[2] = a[3] = 0$ and $a[0] < 4845$. a converges to $[0, 4864)$ after a fast reduction without generating any carry bit.
- *Case 2*: $2^{255} - 19 <= a < 2^{255}$, with the elements satisfying $a[1] = a[2] = a[3] = a[4] = 2^{51} - 1$ and $2^{51} - 19 <= a[0] < 2^{51}$. Under this case a is required to subtract q to get fully reduced.

Since the above two cases never occur at the same time, we perform both the two reductions for side-channel resistance as shown in Figure 16.

4.2 Protocol Stack

Based on the primitives introduced above, we construct a protocol stack including SRP and SPAKE2+ across client-side and server-side.

4.2.1 Client Side. As we have discussed before, offline computing is infeasible on the client-side because of the mechanism of JavaScript. Initial vectors and look-up tables will be generated on instantiating of the owner object. The pre-computed table of a number or an EC point in *pake.js* is generated on the fly and then cached for reused in subsequent exponentiation or point multiplications. We make the group generator a member of the relevant object, i.e. *srp* and *spake2p*, hereafter the table takes effect in later authentications until the object is destroyed. According to Figure 1, the immediate value g^x in step 5 can also be reused, if we make it a member likewise.

4.2.2 Server Side. RFC5054 specifies prime groups with the representative modulus of 1024-bits to 8192-bits in length for SRP, and in this work, we adopt the first three whose modulus bit lengths are 1024, 1536 and 2048 respectively. According to Table 1, there are three modular exponentiation operations on the server-side during a single execution of SRP, including a fixed-base one. For an unknown base, we pre-compute the fixed window table for its exponentiation and determine the optimal window size as 6 after repeated trials for the three groups. For a fixed base such as a group generator, we use pre-computed tables for acceleration and set the window size w to 8. Consequently, the offline files are 5MB, 11.25MB and 20MB respectively. A slightly larger w brings limited performance improvement but an exponential increase of memory occupation. However, if w doubles, the memory footprint will reach 640MB, 1440MB and 2560MB, which are unacceptable for either RAM or global memory on GPUs. For large integers of the three lengths, we conduct calculations with $\lfloor l/1024 \rfloor \times 4$ threads on GPUs through trials, where l is the modulus length in bits of a designated group, that is, 4 threads for 1024-bits and 1536-bits while 8 threads for 2048-bits.

The verifier of SPAKE2+ consists of w_0 and $L = [w_1]P$, where w_0 and w_1 are elements of $GF(p25519)$ derived from user's password and P is a generator of Edwards25519 group. In our framework, the Web server sends tuple (w_0, L, X) to PAKE-Server and gets a response with (Y, Z, V). As M, N and w_0 are fixed for an individual user, we compute the point multiplications $[w_0]N$ and $[w_0]M$ in Step 5 and Step 6 of Figure 2 in advance and replace the original verifier (w_0, L) in database with $(w_0, [w_0]M, [w_0]N, L)$. Consequently, the tuple sent to PAKE-Server becomes $([w_0]M, [w_0]N, L, X)$ with 32-bytes larger in size than before, and this modification saves two point multiplications at the expense of a 32-bytes increase in the size of each request to PAKE-Server and additional 4 point-decompresses which contain square roots.

4.3 Up-level Optimizations

While the optimized protocol stack helps speed up the execution of a single PAKE instance, reasonable I/O handling and task scheduling will contribute to the overall performance of the system. Figure 7 shows the detailed structure of PAKE-Server.

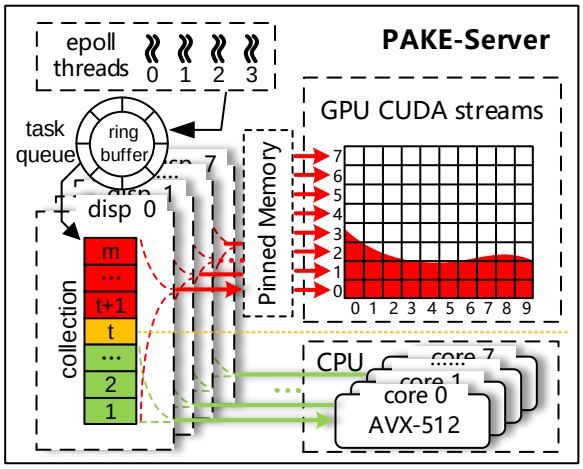


Figure 7: Detailed Structure of PAKE-Server

4.3.1 Asynchronous Computing Routine. While aPAKE service belongs to "computation-bound" programs for its heavy overheads, we adopt an asynchronous way to avoid I/O blocking under highly concurrent requests. More specifically, requests are not processed immediately on arrival, instead, they are collected by a group of dispatching threads for batch processing. This method is widely used in GPU-accelerated servers, where at least a global thread is required for gathering up tasks. However, most of the time the tasks are too little to fully leverage the computing power of GPUs and consequently, an individual request's latency is higher than CPU-based implementations. This is why we provide AVX-accelerated implementations in addition to CUDA programs. The system contains an I/O module and a worker module.

4.3.2 I/O Module. This module consists of multiple epoll threads and dispatchers bridged by a task queue. As a new system call introduced in Linux 2.5.44 to replace the older POSIX *select* [26] and *poll* [25], *epoll* [24] solves the problem of scalable I/O event notification. An epoll thread monitors a socket pool with *epoll* interfaces, sending the IDs of arrived packages to the task queue. A dispatcher (denoted by *disp* in the figure) collects tasks from the queue and routes them to CPU or GPUs according to the task number and dispatching strategy. The two types of threads are not the more the better. Besides the restriction of CPU resources, too many dispatchers also lead to inefficient GPU utilization. The processor we use has 10 physical cores, which means 20 logical cores with hyper-threading enabled. Owing to the excellent behavior of *epoll*, we only keep 4 epoll threads and 2 task queues. The dispatcher number is set to 8 over repeated trials. As our device is equipped with an NVIDIA Titan V card which has 80 streaming multiprocessors (sm), we maintain 10 CUDA streams

for each dispatcher to fully exploit the parallelism of GPUs. With the zero-copy feature enabled since CUDA 2.2, we allocate pinned mapped memory for each stream in advance to avoid expensive host-device memory copy.

4.3.3 Worker Module. Tasks are forwarded to this module for computation finally. For each protocol, we provide two implementation versions, host program based on AVX-512 instructions and CUDA program for GPUs. With the control of dispatchers, we construct a heterogeneous computing model. If the request number is "relatively small", the tasks will be finished by the host program one by one. Otherwise, the tasks will be launched to GPUs and processed by the CUDA program in bulk. According to the specifications of our graphics card listed in Table 2, we schedule the CUDA threads as follows:

- BlocksPerGrid = 2
- GridsPerStream = MaxResidentBlocksPerSM / BlocksPerGrid = 16
- ThreadsPerBlock = MaxResidentThreadsPerSM / MaxResidentBlocksPerSM = 64

Table 2: Specifications of NVIDIA Titan V (Compute Capability 7.0)

Number of Streaming Multiprocessors (SM)	80
Max Resident Threads Per SM	2048
Max Resident Blocks Per SM	32
Max Resident Grids Per Device	128

There ought to be a threshold t in a dispatcher to judge whether the tasks should be sent to GPUs or not. The proper value of t depends on multiple factors, such as computational overheads, optimizations, hardware specifications, etc. In other words, t is an empirical value, and we will determine the optimal t of each protocol implementation through experiments in Section 5.

4.4 Application to Web Systems

The overall framework of our scheme is displayed in Figure 8. The dotted box at the left indicates the border of the Web application, with the blue shading blocks highlighting new components for vendors to integrate.

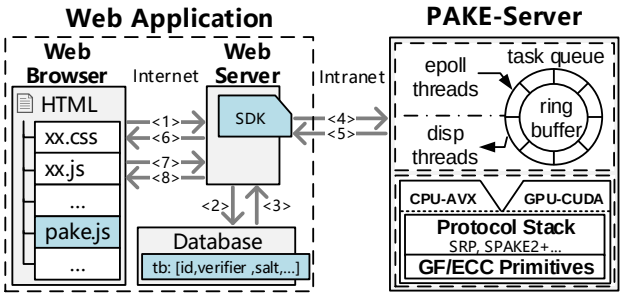


Figure 8: Overall Framework

For Web browsers, vendors only need to insert *pake.js* into their front-end code. Meanwhile, PAKE-Server deployed in Intranet takes over the most costly work of aPAKE computations from Web servers.

In contrast, upgrading Web servers is a bit more complicated. The most troublesome job is database modification. If the target protocol (e.g., SRP) hashes the password together with a random salt that the system hasn't used before, the developers should establish storage for salts. While vendors are suggested to store the salts separately from the passwords in case of leakage, the involvement of salt won't change the structure of a table. The functionalities of server-side aPAKE could be divided into two categories, lightweight operations (e.g., hash, KDF, HMAC), and resource-intensive public-key computations. Since the latter type of operations is highly centralized and computationally expensive, they are offloaded to the PAKE-Server. Web servers are required to integrate the left functions and TCP connection management. In this work, we provide the new functionalities with an easy-to-integrate Software Development Kit (SDK), which acts as middleware.

It is worth mentioning that if the Web servers are equipped with AVX-512-available processors (such as Intel Xeon family based on Skylake microarchitecture), our scheme could be tweaked to a partial plug-in mode by transplanting CPU program to Web servers. Such adjustment further improves performance for minimal tasks.

The numbered arrows in Figure 8 indicate the data flow of authentication. Flow $\langle 1 \rangle$ to $\langle 6 \rangle$ derive the session key while the rest finish key confirmation. In our design, SRP and SPAKE2+ run as Table 3 shows.

Table 3: Transferred Parameters of SRP and SPAKE2+ in Figure 8

	Data Flow	SRP	SPAKE2+
Key Derivation	$\langle 1 \rangle$	ID, A	ID, X
	$\langle 2 \rangle$	(select s, v)	(select $[w_0]M, [w_0]N, L$)
	$\langle 3 \rangle$	s, v	$[w_0]M, [w_0]N, L$
	$\langle 4 \rangle$	A, v	$X, [w_0]M, [w_0]N, L$
	$\langle 5 \rangle$	u, B, S	Y, Z, V
	$\langle 6 \rangle$	u, B, s	Y
Key Confirmation	$\langle 7 \rangle$	M_1	K_{cA}
	$\langle 8 \rangle$	M_2	K_{cB}

5 EVALUATION

In this section, we conduct a series of experiments to reveal the performance of our scheme. The protocols we implement include SRP3 protocol with modulus size 1024-bit, 1536-bit and 2048-bit as well as SPAKE2+ protocol over Edwards25519 curve. For simplicity, we denote SRP implementation of supposed parameters with SRP-l, where l is the modulus bit length, e.g. SRP-1024.

5.1 Basic Configurations

We build the testing environment under LAN as well as WLAN. Besides PAKE-Server and a Web server, the client devices include a wireless mobile phone and a PC connected to the network with a cable. Table 4 and Table 5 give the specifications and configurations of the devices.

Table 4: Server-side Specifications

	Web Server	PAKE-Server
CPU	Intel® Xeon® E5-2697 v2	Intel® Core(TM) i9-7900X
GPU	-	NVIDIA GeForce GTX TITAN V
RAM	8GB	16GB
OS	Linux 4.15.0-64-generic, Ubuntu 16.04.16	Linux 4.15.0-141-generic, Ubuntu 18.01
Network Card	82574L Gigabit Network Connection	Ethernet Connection (2) I219-V
Software	Apache/2.4.18 & PHP 7.0.33	-

Table 5: Clients' Specifications

Product	Xiaomi Laptop Air 13	HUAWEI Mate 30 Pro
CPU	Intel® Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz	HUAWEI Kirin 990 5G
RAM	8GB	8GB
OS	Windows 10 Family 20H2	Android 10

5.2 Determining the Optimal Threshold

As we have mentioned the threshold t for heterogeneous computing in Section 4, we determine its optimal value through experiments here.

5.2.1 Theoretical Model. From the client-side's perspective, the total delay increases with the number of tasks sent to PAKE-Server. Theoretically, the CPU program's delay increases linearly, conforming to $y = k \cdot x$ while the GPU program's delay increases stepwise, which could be concluded as $y = \lceil \frac{x}{B} \rceil \cdot T$. The variable x denotes task number, and y denotes delay. In addition, the parameter T indicates the latency for a single task processed by GPUs, and B implies the maximum amount of tasks that arrive simultaneously, which is decided by the connection number and the socket buffer's capacity for protocol-specific messages.

A proper t shall be abscissa of the only intersection with the two function curves, and hence once the task number exceeds t the GPU always finishes responds faster than the CPU. This requires the slope k not to be less than $2T/B$, and then the inequality $B >= \lceil 2T/k \rceil$ holds. After that, we could get the optimal t by substituting y with T in equation $y = k \cdot x$, and finally $t = T/k$.

Since T could hardly be reduced, the feasible approaches to guarantee the existence and the uniqueness of t include enlarging B by expanding the socket buffer and establishing more connections to PAKE-Server. Our experiment intends to acquire T and the lower limit of B for each protocol.

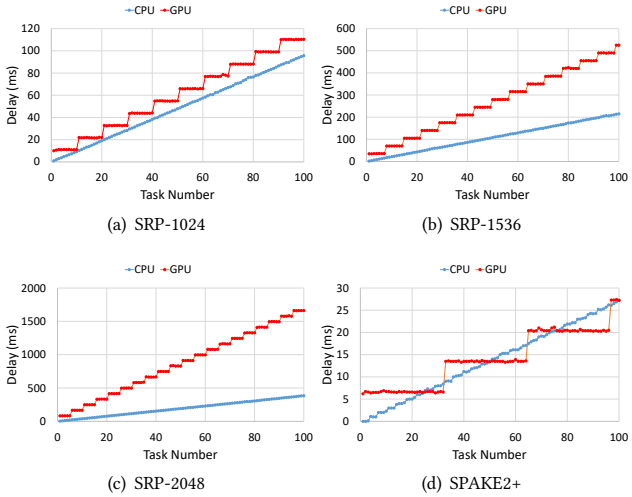
5.2.2 Experiment Setup. Instead of sending requests from browsers, we launch tests on PAKE-Server directly with a client program that could simulate requests with configurable parameters like task amount, connection number. We only maintain one dispatcher for the test to avoid the error resulted from thread competitions.

The function curves of time-delay are shown in Figure 9 and more detailed data is given in Table 11 in Appendix B. Since we adopt a small socket buffer size (4096-bytes) for convenience, the two curves in each diagram do not intersect at an expected point with a small B. However, this won't impact the accuracy of t.

Just as we analyzed above, in each subfigure, the slope k of the CPU curve and the step height T of the GPU curve are determined by the complexity of the target protocol. A heavier computation overhead leads to greater k and T, and thus the four protocols could be sorted by complexity as SRP-2048, SRP-1536, SRP-1024 and SPAKE2+. After the statistical analysis for the experimental result, key parameters of each protocol are given in Table 6. The row *deduced t* means that the value is deduced as $t = \lceil T/k \rceil$, and *empirical t* indicates the task number of the request whose response time firstly exceeds T in Table 11. The coincidence of the two rows implies the theoretical model's correctness.

5.3 Performance Evaluation from Server's Perspective

While multiple metrics are taken into account for performance evaluation of the system, particular concerns are paid to *throughput*



(a) SRP-1024 (b) SRP-1536

(c) SRP-2048 (d) SPAKE2+

Figure 9: Time-delay of a Single Dispatcher

Table 6: Key Parameters

Protocol	SRP-1024	SRP-1536	SRP-2048	SPAKE2+
T	10.997	33.526	83.074	6.788
k	0.957	2.161	3.842	0.269
suggested B ($\lceil 2T/k \rceil$)	≥ 23	≥ 32	≥ 44	≥ 51
deduced t ($\lceil T/k \rceil$)	12	16	22	26
empirical t	12	16	22	26

T: latency of a single task on GPUs k: the slope of CPU delay function
B: max number of tasks a socket buffer contains t: threshold

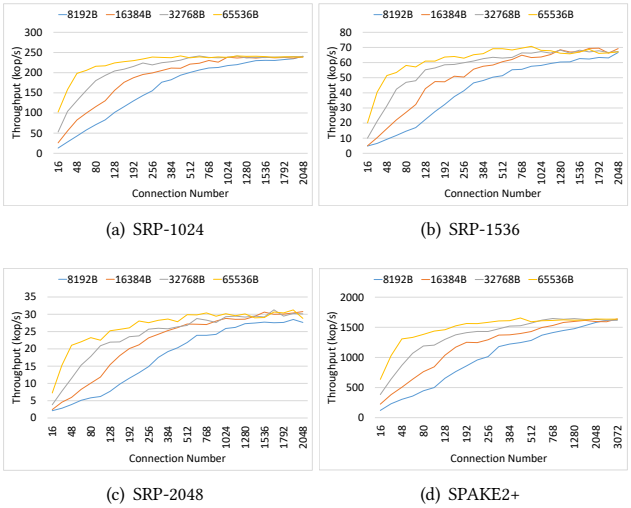
for PAKE-Server, which indicates the tasks processed per second. By testing the maximize *throughput*, we expect to provide references for making the most of PAKE-Server's computing capacity.

5.3.1 Trade-off Considerations. The basic routine for improving *throughput* is to increase c (number of connections to PAKE-Server) and p (number of tasks carried in a roundtrip message) by establishing more TCP connections with a larger socket buffer. However, larger c or p leads to higher resource consumption and logical complexity. A proper configuration for c and p helps the Web system harvest the best performance without obvious waste of resources. While the p value of a protocol is determined after compilation, the c value could be controlled by the vendors according to the scale of their server cluster. In our design, vendors are suggested to

(1) select the proper socket buffer size according to the current size and the probable size in the future of their server cluster, and
(2) establish an appropriate number of TCP connections to PAKE-Server in each Web server to ensure that it can achieve the ultimate *throughput* exactly.

5.3.2 Experiment Setup. In order to provide reasonable c and p values for vendors, we mount stress testing with different connection numbers and socket buffers size of 8192-bytes, 16384-bytes, 32768-bytes and 65536-bytes respectively. The detailed results are shown in Table 12 and Table 13 while intuitive results are drawn in Figure 10.

With p determined by the socket buffer size, *throughput* increases with c until the peak value. Vendors should ensure that



(a) SRP-1024 (b) SRP-1536

(c) SRP-2048 (d) SPAKE2+

Figure 10: Throughput of PAKE-Server

c reaches the inflection point exactly in the figures. Oversized c will not bring any performance improvement, instead, it leads to resource wastes of Web servers. Taking SPAKE2+ as an example, suppose that the Web server cluster contains 64 servers. According to Table 11, by adopting a 65536-bytes socket buffer, PAKE-Server could get fully loaded with only 5 TCP connections to each Web server. As long as the cluster gets expanded to no more than 320 servers in the future, vendors could maintain the best performance of the system by reducing TCP connections in each Web server to a proper number.

The abscissa of the inflection point of each curve in the figures represents the optimal number of connections under the current buffer size. Table 7 gives the maximum *throughput* of each protocol and the recommended c value for each socket buffer size is listed in Table 8.

Table 7: Peak Throughput

Protocol	SRP-1024	SRP-1536	SRP-2048	SPAKE2+
Peak Throughput (kop/s)	241.536	70.740	30.155	1654.947

Table 8: Suggested Connection Number

Socket Buffer Size	Protocol			
	SRP-1024	SRP-1536	SRP-2048	SPAKE2+
8192B	2048	2048	2560	2560
16384B	1024	1152	1408	1280
32768B	512	768	1024	640
65536B	256	320	512	320

5.4 Performance Evaluation from Clients' Perspective

In this subsection, we conduct tests for PAKE protocols from the client side. In comparison with PAKE-Server, we focus on user experience with the metric *latency* which implies the time delay from the user submitting the form to the moment session key is established.

5.4.1 Preparation. We set up a Web server with a simple PHP website on Apache server, and launch requests from four mainstream PC browsers and two mobile browsers. The detailed information of the browsers is listed in Table 9. Our implementations conform to RFC2945 and CFRG document [42], and the KDF function used for SPAKE2+ is PBKDF2 with iteration set to 1000.

Table 9: Versions of Testing Platforms

Platform	Version
Chrome	90.0.4430.93 (Official Build) (64-bit)
Firefox	88.0 (64-bit)
IE	11
Microsoft Edge	90.0.818.51 (64-bit)
Huawei Browser	11.0.8.301
Firefox for Android	88.13 (Build #2015808649)

5.4.2 Experiment Setup. In order to observe the detailed time consumption on the call path, we collect timestamps of several key nodes as shown in Figure 11, and then get the delays of desired phases.

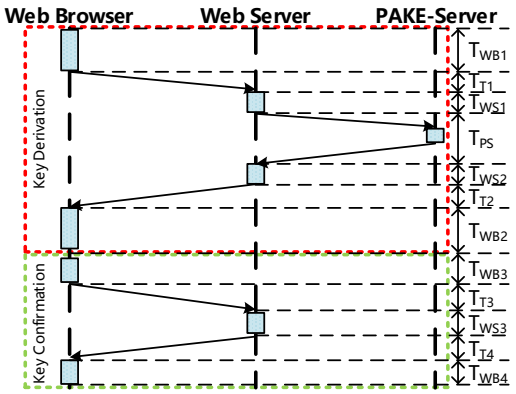


Figure 11: Time Slots, the subscript "$WB*$" means "Web Browser", "$T*$" means "Transmission", "$WS*$" means "Web Server", and "PS" means "PAKE-Server"

We launch tests during idle-hours as well as busy-hours of PAKE-Server. Considering the disturbance caused by the unstable network (especially WLAN), we sample the transmission delay T_{T1}, T_{T2}, T_{T3} and T_{T4} to diminish the infect of network variations. Influenced by the out-sync clocks between the client and the server, a single transmission delay is inaccurate (sometimes even be negative). However, the sum of them can offset this error, and we denote it with T_T. The testing results are shown in Table 14-17 in Appendix B.

To reveal the performance gap between PAKE schemes and a pure hash-based authentication, we test the latter's delay and list the results in Table 18. The hash function we adopted is Bcrypt. Figure 12 compares the *latency* during idle hours on the testing platforms with histograms. Although the hash-based approach seems to be much faster than PAKE schemes, its workloads are centralized in Web servers, leading to a rapid increase of *latency* with the concurrency up.

Unsurprisingly, the outdated kernel of IE results in the highest latency among the testing platforms. All the protocols could be finished within 3 seconds except for SRP-2048 on IE. Influenced

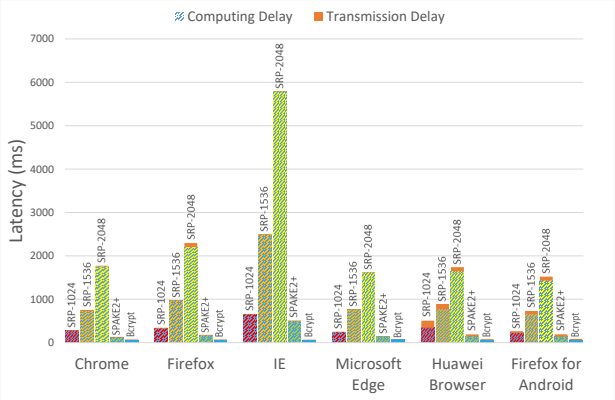


Figure 12: Latency during Idle-hours

by the natural shortage of wireless networks, the two mobile browsers receive higher transmission delays than the desktop ones. While a PAKE scheme achieves different performances on various browsers, the hash-based authentication obtains a stable delay since the hash operation is conducted on the server-side.

5.4.3 Result Analysis. In the low concurrency scenes, T_{PS} only shares a small proportion (seldom exceeds 1%) of the entire process. On the contrary, most time consumption occurs in Web browsers for the natural performance shortcomings of JavaScript and consequently, other factors including browser kernel, crypto parameters (such as the iteration number of PBKDF2) also impact the client-side's performance significantly, and thus latency of the same protocol varies on different platforms. When PAKE-Server is fully loaded, the user could acutely feel the increase of delay. According to the result, T_{PS} under the busy state is 10x to 100x of that during idle time. Although the latency of a PAKE authentication seems to be times of a hash-based one, the bottleneck lies in browsers instead of Web servers. According to Table 18 and T_{WS*} in Table 14-17, our scheme takes over almost all the burdens of Web servers in comparison with traditional hash-based approach, and hence the system's throughput increases by a large margin. It is noticeable that SPAKE2+ keeps latency lower than 0.4s even when facing up with over 1,600,000 requests per second on most platforms.

6 CONCLUSION

In this work, we propose Heterogeneous-PAKE, a practical Web framework for the application of PAKE protocols. Our scheme also gives the vendors instructions about upgrading existing Web systems for PAKE authentication. To dispel people's misgiving about the feasibility of PAKE schemes in a real-world scene caused by their heavy computational overheads, we provide high-speed implementations of two representative PAKE protocols, SRP and SPAKE2+. The system achieves comprehensive performance comparable with hash-based authentications by integrating state-of-the-art research results into a heterogeneous computing module supported by GPUs and CPU with vector instructions. For PAKE protocols, this work bridges the gap between their theoretical research and the real-world deployment and provides a reference for industrialization.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China under Grant 61902392 and in part by CCF-Tencent Open Fund under Grant RAGR20210131.

REFERENCES

[1] 2019. Facebook stored hundreds of millions of passwords in plain text. https://www.theverge.com/2019/3/21/18275837/facebook-plain-text-password-storage-hundreds-millions-users
[2] 2019. *Google stored some passwords in plain text for fourteen years–Only affects some G Suite customers.* Technical Report. https://www.theverge.com/2019/5/21/18634842/google-passwords-plain-text-g-suite-fourteen-years
[3] Michel Abdalla and David Pointcheval. 2005. Simple password-based encrypted key exchange protocols. In *Cryptographers' track at the RSA conference.* Springer, 191–208.
[4] Steven Michael Bellovin and Michael Merritt. 1992. Encrypted key exchange: Password-based protocols secure against dictionary attacks. (1992).
[5] F. Callegati, W. Cerroni, and M. Ramilli. 2009. *Man-in-the-Middle Attack to the HTTPS Protocol.* Man-in-the-Middle Attack to the HTTPS Protocol.
[6] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science.* IEEE, 136–145.
[7] Marius Cornea. 2015. Intel AVX-512 instructions and their use in the implementation of math functions. *Intel Corporation* (2015).
[8] Jiankuo Dong, Fangyu Zheng, Juanjuan Cheng, Jingqiang Lin, Wuqiong Pan, and Ziyang Wang. 2018. Towards high-performance X25519/448 key agreement in general purpose GPUs. In *2018 IEEE Conference on Communications and Network Security (CNS).* IEEE, 1–9.
[9] Jiankuo Dong, Fangyu Zheng, Niall Emmart, Jingqiang Lin, and Charles Weems. 2018. sDPF-RSA: Utilizing floating-point computing power of GPUs for massive digital signature computations. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 599–609.
[10] Niall Emmart, Fangyu Zheng, and Charles Weems. 2018. Faster modular exponentiation using double precision floating point arithmetic on the GPU. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH).* IEEE, 130–137.
[11] Lili Gao, Fangyu Zheng, Niall Emmart, Jiankuo Dong, Jingqiang Lin, and Charles Weems. 2020. DPF-ECC: Accelerating Elliptic Curve Cryptography with Floating-Point Computing Power of GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 494–504.
[12] Jeffrey Goldberg. 2017. *Three layers of encryption keeps you safe when SSL/TLS fails.* Technical Report. https://blog.1password.com/three-layers-of-encryption-keeps-you-safe-when-ssl/tls-fails/
[13] IEEE P1363 Working Group et al. 2003. Standard specifications for password-based public-key cryptographic techniques. *IEEE P1363. 2/D11* (2003).
[14] Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. 2008. Twisted Edwards curves revisited. In *International Conference on the Theory and Application of Cryptology and Information Security.* Springer, 326–343.
[15] Apple Inc. 2021. *Apple Platform Security.* Technical Report. https://manuals.info.apple.com/MANUALS/1000/MA1902/en_US/apple-platform-security-guide.pdf
[16] D. JABLON. 1999. B-SPEKE. *Integrity Sciences White Paper* (1999). https://ci.nii.ac.jp/naid/10010452557/en/
[17] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. 2018. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, 456–486.
[18] Hwancheol Jeong, Sunghoon Kim, Weonjong Lee, and Seok-Ho Myung. 2012. Performance of SSE and AVX instruction sets. *arXiv preprint arXiv:1211.0820* (2012).
[19] William Kahan. 1996. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE* 754, 94720-1776 (1996), 11.
[20] Cameron F Kerry and Charles Romine Director. 2013. FIPS PUB 186-4 federal information processing standards publication digital signature standard (DSS). (2013).
[21] Taekyoung Kwon. 2001. Authentication and Key Agreement via Memorable Password.. In *NDSS.*
[22] K. Leboeuf, R. Muscedere, and M. Ahmadi. 2013. A GPU implementation of the Montgomery multiplication algorithm for elliptic curve cryptography. *IEEE* (2013).
[23] Philip MacKenzie. 2002. The PAK suite: Protocols for password-authenticated key exchange. In *IEEE P1363. 2.* Citeseer.
[24] Michael Kerrisk. 2021. epoll(7) - Linux manual page. https://man7.org/linux/man-pages/man7/epoll.7.html [Online; accessed 16-June-2021].
[25] Michael Kerrisk. 2021. poll(2) - Linux manual page. https://man7.org/linux/man-pages/man2/poll.2.html [Online; accessed 16-June-2021].
[26] Michael Kerrisk. 2021. select(2) - Linux manual page. https://man7.org/linux/man-pages/man2/select.2.html [Online; accessed 16-June-2021].
[27] Montgomery and L. Peter. 1985. Modular multiplication without trial division. *Math. Comp.* 44, 170 (1985), 519–519.
[28] Kathleen Moriarty, Burt Kaliski, and Andreas Rusch. 2017. Pkcs# 5: Password-based cryptography specification version 2.1. *Internet Eng. Task Force (IETF)* 8018 (2017), 1–40.

[29] Junghyun Nam, Juryon Paik, H-K Kang, Ung Mo Kim, and Dongho Won. 2009. An off-line dictionary attack on a simple three-party key exchange protocol. *IEEE Communications Letters* 13, 3 (2009), 205–207.
[30] Stuart Oberman, Greg Favor, and Fred Weber. 1999. AMD 3DNow! technology: Architecture and implementations. *IEEE Micro* 19, 2 (1999), 37–48.
[31] Wuqiong Pan, Fangyu Zheng, Yuan Zhao, Wen-Tao Zhu, and Jiwu Jing. 2016. An efficient elliptic curve cryptography signature server with GPU acceleration. *IEEE Transactions on Information Forensics and Security* 12, 1 (2016), 111–122.
[32] Alex Peleg and Uri Weiser. 1996. MMX technology extension to the Intel architecture. *IEEE micro* 16, 4 (1996), 42–50.
[33] Colin Percival and Simon Josefsson. 2016. The scrypt password-based key derivation function. *IETF Draft URL: http://tools. ietf. org/html/josefsson-scrypt-kdf-00. txt (accessed: 30.11. 2012)* (2016).
[34] Mark Pilgrim. 2010. Dive into HTML5. *URL: http://diveintohtml5.info/index.html* (2010).
[35] Niels Provos and David Mazieres. 1999. Bcrypt algorithm. In *USENIX*.
[36] Venu Gopal Reddy. 2008. Neon technology introduction. *ARM Corporation* 4, 1 (2008).
[37] Eric Rescorla and Tim Dierks. 2018. The transport layer security (TLS) protocol version 1.3. (2018).
[38] Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.
[39] Nicolas Serrano, Hilda Hadan, and L Jean Camp. 2019. A complete study of PKI (PKI's Known Incidents). *Available at SSRN 3425554* (2019).
[40] SeongHan Shin and Kazukuni Kobara. 2012. Efficient Augmented Password-Only Authentication and Key Exchange for IKEv2. RFC 6628. https://doi.org/10.17487/RFC6628
[41] Emily Stark, Michael Hamburg, and Dan Boneh. 2009. Symmetric cryptography in javascript. In *2009 Annual Computer Security Applications Conference*. IEEE, 373–381.
[42] Tim Taubert and Christopher A. Wood. 2020. SPAKE2+, an Augmented PAKE. https://datatracker.ietf.org/doc/html/draft-bar-cfrg-spake2plus-02 Work in Progress.
[43] Abi Tyas Tunggal. 2021. *The 56 Biggest Data Breaches (Updated for 2021)*. Technical Report. https://www.upguard.com/blog/biggest-data-breaches
[44] Henk C. A. van Tilborg and Sushil Jajodia (Eds.). 2011. *Fixed Window Exponentiation*. Springer US, Boston, MA, 482–482. https://doi.org/10.1007/978-1-4419-5906-5_1168
[45] Y. Wang. 2001. IEEE P1363.2 Submission / D2001-06-21,. P1363.2-ecsrp-06-21.doc. (21 June 2001).
[46] Thomas Wu. 2002. Srp-6: Improvements and refinements to the secure remote password protocol. *http://srp. stanford. edu/srp6. ps* (2002).
[47] Thomas D Wu et al. 1998. The Secure Remote Password Protocol.. In *NDSS*, Vol. 98. Citeseer, 97–111.

A AVX-512 CODES

```
1  #define ROUND _MM_FROUND_TO_ZERO|_MM_FROUND_NO_EX
2
3  dpf_full_product(__m512d a_samples, double b)
4  {
5      __m512d hi, lo, tmp, b_samples = vpbroadcastq (b),
6          _c1 = vbroadcastpd (0x4670000000000000ull),
7          _c2 = vbroadcastpd (0x4670000000000001ull);
8
9      hi = vfmaddpd (a_samples, b_samples, _c1, ROUND);
10     tmp = vpsubpd (_c2, hi);
11     lo = vfmaddpd (a_samples, b_samples, tmp, ROUND);
12     return (hi, lo);
13 }
```

$vpbroadcastq$ is abbreviated for sequence of instructions which broadcasts DPF (64-bit) a to all elements of dst
$vfmaddpd$ here denotes the combination of $vfmadd132pd$, $vfmadd213pd$ and $vfmadd231pd$

Figure 13: dpf_full_product for AVX-512

```
1  dpf_full_product(__m512i a_samples, uint64_t b)
2  {
3      __m512i hi, lo, _zero = vpbroadcastq (0),
4      b_samples = vbroadcastq (b)
5      hi = vpmadd52huq (a_samples, b_samples, _zero);
6      lo = vpmadd52luq (a_samples, b_samples, _zero);
7      return (hi, lo);
8  }
```

$vpbroadcastq$ is abbreviated for sequence of instructions which broadcasts DPF (64-bit) a to all elements of dst

Figure 14: dpf_full_product for AVX-512 with IFMA

Table 10: Preseted Variables

Symbol	ZMM Elements (LE)
_ones	[1, 1, 1, 1, 1, 1, 1, 1]
_19	[19, 19, 19, 19, 19, 19, 19, 19]
_q	$[2^{51}-19, 2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{51}-1, 0, 0, 0]$
_mask51	$[2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{51}-1]$
_mask	$[2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{51}-1, 2^{52}-1]$

```
1  __m512i gf25519_add(__m512i a)
2  {
3      __mmask8 generate, propagate;
4      __m512i c = vpaddq (0x1f, a, b), t
5      /* carry-predicting */
6      t = vpandq (c, 0xf, c, _mask51);
7      propagate = vpcmpuq (t, _mask, EQ);
8      generate = ((generate<<1)+propagate) ∧ propagate;
9      c = vpaddq (t, generate, t, _ones);
10     /* fast reduction */
11     t = valignq (_zero, c, 4);
12     t = vpsrlq (t, 51);
13     t = vpmullq (t, _19);
14     c = vpandq (c, _mask51);
15     c = vpaddq (c, t);
16     /* carry resolve */
17     t = vpaddq (_mask, _one);
18     generate = vpcmpuq (c, t, NLT);
19     generate ⊕ = (generate << 1 & 2) + generate;
20     t = vpaddq (c, generate, c, _ones);
21     return vpandq (t, _mask);
22 }
```

Figure 15: Modular Addition over $GF(p25519)$

```
1  __m512i gf25519_final_red(__m512i a)
2  {
3      __mmask8 generate = vpcmpuq (0x1f, a, _mask51, NLE);
4      __m512i t = vpandq (a, _mask51);
5      a = vpaddq (t, generate>>4, t, _19);
6      generate = vpcmpuq (0x1f, a, _q, NLT);
7      return vpsubq (a, (generate+1>>5)*0x1f, a, _q);
8  }
```

Figure 16: Final Reduction over $GF(p25519)$

B EXPERIMENTAL RESULTS

Table 11: Time-delay of a Single Dispatcher

Tasks	SRP-1024		SRP-1536		SRP-2048		SPAKE2+		Tasks	SRP-1024		SRP-1536		SRP-2048		SPAKE2+	
	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU		CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
1	1.0	10.1	2.0	35.0	3.8	82.4	0.0	6.2	51	48.7	65.9	111.0	279.5	195.3	913.4	14.0	13.6
2	2.0	10.7	4.0	34.8	7.8	83.1	0.0	6.7	52	49.9	65.8	112.6	280.0	199.2	911.2	14.1	13.5
3	3.0	11.2	6.8	35.3	11.7	82.9	0.1	6.6	53	50.7	65.8	114.5	280.0	205.5	913.9	14.3	13.5
4	3.8	11.0	8.7	35.3	15.3	83.4	1.1	6.4	54	51.8	66.0	116.1	280.3	207.3	910.9	15.0	13.5
5	4.8	11.2	11.0	36.3	19.6	83.0	1.0	6.5	55	52.9	65.8	119.1	280.1	210.2	911.2	15.3	13.5
6	5.6	11.1	13.0	35.3	23.7	166.2	1.0	6.5	56	53.9	65.9	120.9	280.4	214.8	997.4	15.3	13.3
7	6.5	11.3	15.2	35.8	27.5	166.2	2.0	6.5	57	54.6	66.0	124.3	315.6	218.8	997.5	15.3	13.4
8	7.3	10.9	17.6	70.1	31.1	166.5	2.0	6.7	58	55.4	65.7	125.1	315.5	222.6	996.8	15.9	13.5
9	8.3	11.0	19.7	70.0	34.9	166.2	2.0	6.9	59	56.5	66.0	127.1	315.3	226.5	996.9	16.1	13.5
10	9.1	11.1	21.7	70.0	38.8	166.0	2.3	6.7	60	57.5	65.9	128.9	315.2	229.8	996.1	16.1	13.9
11	10.2	22.0	23.8	70.0	42.5	249.2	3.0	6.7	61	58.4	76.9	132.1	315.5	233.9	1080.4	16.1	13.5
12	11.2	21.8	26.1	70.2	46.4	249.4	3.0	6.6	62	59.6	76.9	133.8	315.5	237.5	1078.3	16.5	13.5
13	12.3	21.8	28.4	70.0	49.8	249.5	3.0	6.6	63	60.5	77.1	136.9	315.0	240.6	1080.7	17.0	13.5
14	13.4	22.1	30.2	70.3	54.0	249.5	3.8	6.5	64	60.6	77.0	137.8	350.3	245.1	1076.2	17.1	13.6
15	14.0	21.8	32.4	105.0	57.5	249.3	4.0	6.7	65	62.3	76.9	139.5	350.1	249.9	1080.4	17.5	20.4
16	15.0	21.7	35.0	105.1	62.0	332.0	4.0	6.6	66	62.8	76.8	142.2	350.3	253.8	1158.8	17.9	20.5
17	16.0	21.6	36.6	105.1	65.8	332.3	4.2	6.7	67	64.1	77.1	145.3	349.4	258.2	1165.0	18.2	20.3
18	17.0	21.6	38.8	105.0	69.4	334.3	4.9	6.6	68	65.0	78.7	146.6	350.5	260.2	1163.2	18.3	20.4
19	18.1	22.2	41.5	104.9	73.7	333.6	5.0	6.6	69	66.0	78.1	149.1	350.2	264.4	1160.8	19.0	21.0
20	19.0	22.0	43.0	105.5	77.8	332.3	5.0	6.6	70	66.9	77.3	150.0	350.6	268.2	1163.2	19.2	20.6
21	20.0	32.8	45.4	105.1	81.0	415.4	5.4	6.5	71	67.6	87.9	152.6	384.2	272.6	1244.0	19.1	20.4
22	21.1	32.5	47.5	140.0	85.3	416.8	6.0	6.6	72	68.7	88.1	154.8	385.0	276.9	1244.9	19.5	20.4
23	22.1	32.8	49.9	140.0	88.2	414.9	6.0	6.7	73	70.3	88.1	157.4	385.1	280.1	1244.8	19.9	20.4
24	23.1	32.6	52.0	140.3	92.6	417.3	6.3	6.5	74	70.9	88.0	159.7	385.8	284.9	1244.0	20.2	21.0
25	24.1	32.9	55.2	140.2	96.5	417.3	6.9	6.6	75	71.5	88.1	161.4	385.6	287.7	1245.4	20.4	21.2
26	25.0	32.8	57.7	140.5	100.2	498.3	7.3	6.7	76	73.1	88.0	163.7	385.5	293.9	1325.5	20.4	20.4
27	25.7	32.9	60.1	140.0	103.7	499.1	7.0	6.7	77	74.1	88.1	166.3	385.3	297.1	1328.9	20.6	20.3
28	26.8	32.6	60.7	139.8	107.8	499.3	7.3	6.7	78	75.9	88.0	167.3	420.7	298.2	1329.7	21.0	20.3
29	27.8	32.9	63.1	175.0	111.9	500.1	7.9	6.4	79	76.2	88.0	170.4	421.7	301.8	1327.3	21.6	20.5
30	28.3	32.8	65.0	174.9	115.0	498.7	8.0	6.6	80	76.4	88.1	174.0	423.9	306.0	1328.2	21.9	20.5
31	29.7	43.6	66.9	174.9	119.6	580.9	8.0	6.7	81	77.8	99.3	174.6	420.4	309.2	1409.9	21.9	20.3
32	30.5	44.1	69.4	175.3	123.6	582.2	8.4	6.6	82	78.3	99.1	176.5	420.5	313.2	1412.6	22.2	20.4
33	31.4	43.8	71.7	175.5	127.0	581.6	9.0	13.5	83	79.3	99.0	178.8	420.1	319.3	1414.3	22.2	20.4
34	32.6	43.9	73.1	175.1	130.8	581.9	9.1	13.6	84	80.4	99.2	181.9	420.1	322.6	1411.2	23.0	20.3
35	33.4	43.8	76.0	175.2	135.0	588.1	9.0	13.5	85	81.4	99.1	183.5	455.2	326.9	1412.6	23.0	20.7
36	34.3	43.9	78.6	209.9	138.7	667.2	10.0	13.5	86	82.4	99.1	186.7	454.4	329.4	1495.1	23.2	20.4
37	35.2	43.8	80.3	210.1	142.9	664.4	10.2	13.5	87	83.3	99.0	188.5	455.2	335.0	1497.1	23.3	20.4
38	36.4	43.8	81.7	210.0	146.7	664.3	10.3	13.6	88	84.1	99.0	190.1	455.4	337.8	1497.3	24.0	20.4
39	37.0	43.8	84.3	210.3	149.6	664.0	10.4	13.3	89	85.3	99.0	192.5	454.4	340.2	1494.5	24.3	20.4
40	38.2	44.1	86.4	210.3	153.4	666.1	11.2	13.5	90	86.4	99.1	194.5	454.7	344.7	1494.1	24.2	20.3
41	39.4	54.8	88.7	210.1	157.7	746.9	11.1	13.5	91	87.4	110.1	196.4	455.2	350.0	1579.1	24.3	20.4
42	39.9	55.0	90.6	210.3	161.4	749.2	11.3	13.5	92	88.2	110.3	198.1	490.0	354.3	1576.4	25.2	20.3
43	41.3	55.0	92.8	245.2	165.2	748.4	11.8	13.6	93	89.3	110.2	201.6	490.2	357.7	1577.0	25.1	20.3
44	42.1	55.0	95.2	245.2	168.9	747.7	12.0	13.4	94	89.6	110.3	204.8	490.5	360.6	1583.5	25.3	20.5
45	43.1	54.8	97.0	244.9	172.4	748.7	12.1	13.6	95	91.0	110.3	207.2	489.2	365.8	1576.7	25.7	20.3
46	44.1	54.8	99.5	244.9	176.7	833.3	12.3	13.4	96	91.9	110.0	206.2	490.6	367.9	1662.3	26.2	20.5
47	44.9	54.7	101.6	245.4	181.1	836.9	12.9	13.7	97	93.0	110.3	208.6	490.3	371.8	1660.1	26.1	27.3
48	46.1	54.9	103.5	245.4	184.6	828.6	12.9	13.5	98	93.8	110.2	210.6	489.7	377.3	1661.8	26.5	27.3
49	46.7	54.8	105.8	245.1	187.6	831.2	13.1	13.5	99	94.8	110.3	213.3	525.2	381.9	1660.5	26.9	27.4
50	47.9	54.9	108.5	279.7	192.0	828.5	13.5	13.4	100	95.6	110.3	214.9	525.2	383.9	1662.0	27.3	27.2

Table 12: Throughput of a SRP (kops/s)

Connection Number	SRP-1024				SRP-1536				SRP-2048			
	Socket Buffer Size (B)				Socket Buffer Size (B)				Socket Buffer Size (B)			
	8192	16384	32768	65536	8192	16384	32768	65536	8192	16384	32768	65536
16	13.110	26.041	53.043	102.469	4.844	4.988	10.087	20.163	2.116	2.433	3.832	7.245
32	28.213	55.526	104.122	158.175	6.554	10.373	21.236	40.458	2.828	4.567	7.748	15.239
48	42.896	82.644	130.609	198.172	9.279	16.303	31.318	51.423	3.898	5.948	11.462	21.036
64	57.935	99.859	156.257	205.775	11.847	22.127	42.441	53.463	5.116	8.287	15.293	22.043
80	71.050	115.669	180.343	215.866	14.663	27.112	46.935	58.104	5.840	10.083	17.814	23.241
96	82.964	130.391	193.188	217.049	17.034	32.271	48.021	57.142	6.213	11.849	20.900	22.491
128	101.578	156.214	204.262	224.135	22.367	42.776	55.245	60.979	7.704	15.414	21.934	25.228
160	115.660	176.070	208.573	227.283	27.634	47.453	56.451	60.986	9.793	18.039	22.014	25.645
192	130.041	187.414	215.543	229.813	32.250	47.312	58.553	63.641	11.512	20.076	23.521	26.081
224	143.297	195.442	223.900	233.967	37.490	50.982	58.775	64.030	13.086	21.172	23.755	28.047
256	155.111	199.325	219.218	238.710	41.380	50.443	59.906	62.923	14.875	23.190	25.699	27.574
320	176.322	205.257	224.706	237.884	46.580	55.637	61.043	65.185	17.526	24.225	25.956	28.252
384	182.363	211.364	227.113	237.109	48.132	57.561	62.574	65.903	19.201	25.294	25.796	28.626
448	193.693	210.838	231.013	241.536	50.334	58.365	63.477	69.156	20.302	26.140	26.372	27.846
512	200.456	221.170	237.861	236.871	51.329	60.596	62.970	69.156	21.843	27.175	26.775	29.886
640	206.560	223.406	241.356	239.210	55.256	62.064	63.410	68.324	23.913	27.118	28.776	29.819
768	211.635	230.232	237.844	237.046	55.530	64.898	66.406	69.535	23.919	27.029	28.304	30.396
896	212.862	226.003	236.943	239.088	57.570	63.319	66.248	70.740	24.190	28.046	27.625	29.441
1024	217.467	238.524	237.951	237.864	58.094	63.704	67.267	68.112	25.892	28.815	29.292	30.155
1152	219.878	236.314	241.227	241.468	59.407	65.329	66.248	67.834	26.206	28.531	29.475	29.580
1280	225.054	238.249	236.436	240.509	60.368	68.467	68.262	66.184	27.283	28.576	29.198	30.096
1408	229.814	236.050	237.765	240.726	60.466	67.011	66.248	65.806	27.467	29.393	29.475	28.968
1536	230.598	238.321	237.439	239.829	62.644	67.143	68.199	66.950	27.738	30.631	29.221	29.070
1664	230.325	236.821	238.680	237.799	62.350	69.362	67.063	68.749	27.560	29.998	31.291	30.661
1792	232.507	237.646	238.216	239.489	63.386	69.527	67.785	66.105	27.640	30.051	29.475	30.395
1920	234.596	236.108	239.904	239.037	63.088	66.248	66.656	66.410	28.497	30.374	30.230	31.260
2048	240.823	239.421	239.292	238.537	66.690	69.230	67.343	67.113	27.643	30.783	30.119	28.811

Table 13: Throughput of a SPAKE2+ (kops/s)

Connection Number	Socket Buffer Size (B)				Connection Number	Socket Buffer Size (B)			
	8192	16384	32768	65536		8192	16384	32768	65536
16	120.353	224.465	383.979	635.494	384	1221.913	1375.088	1520.537	1610.178
32	230.379	381.475	639.150	1028.874	448	1246.535	1396.733	1525.923	1654.947
48	304.591	504.489	868.392	1307.136	512	1282.350	1430.174	1570.170	1592.586
64	358.031	637.824	1068.282	1332.777	640	1369.822	1496.873	1619.466	1609.318
80	445.980	763.463	1190.195	1382.840	768	1412.291	1530.439	1646.407	1613.967
96	499.579	843.888	1204.254	1437.757	1024	1447.825	1580.958	1633.715	1623.981
128	653.698	1036.103	1300.162	1459.896	1280	1476.935	1597.757	1644.595	1617.428
160	764.687	1170.432	1374.894	1524.879	1536	1528.371	1615.262	1630.156	1621.012
192	859.364	1251.128	1411.358	1562.808	2048	1580.928	1595.708	1632.368	1638.092
224	957.568	1246.085	1431.111	1561.087	2560	1614.805	1592.281	1617.940	1635.665
256	1012.930	1292.871	1430.446	1582.141	3072	1628.554	1640.023	1615.585	1636.126
320	1179.537	1371.166	1475.133	1605.939					

Table 14: Latency of SRP-1024 during Idle/Busy-Hours

	T_{WB1} (ms)	T_{WS1} (ms)	T_{PS} (ms)	T_{WS2} (ms)	T_{WB2} (ms)	T_{WB3} (ms)	T_{WS3} (ms)	T_{WB4} (ms)	T^*_{total} (ms)	T_{total} (ms)
Chrome	125 / 132	0 / 0	1 / 228	0 / 0	145 / 137	0 / 0	0 / 0	0 / 0	271 / 497	284 / 512
Firefox	150 / 154	0 / 0	1 / 191	0 / 0	177 / 187	0 / 0	0 / 0	0 / 0	328 / 532	343 / 548
IE	274 / 283	0 / 1	1 / 215	0 / 0	363 / 344	0 / 1	0 / 0	0 / 0	638 / 844	648 / 853
Microsoft Edge	109 / 111	0 / 0	1 / 213	0 / 0	133 / 134	0 / 1	0 / 0	0 / 0	243 / 459	252 / 469
Huawei Browser	170 / 116	0 / 0	1 / 201	0 / 0	175 / 181	1 / 0	0 / 0	0 / 0	347 / 498	223 / 264
Firefox for Android	99 / 100	0 / 0	2 / 176	0 / 0	121 / 142	1 / 0	0 / 0	0 / 0	223 / 264	418 / 472

"Busy-Hours" means reaching max throughput (240Kop/s) here
$T^*_{total} = T_{T1} - T_{T4}$, indicating total time without transmission delay

Table 15: Latency of SRP-1536 during Idle/Busy-Hours

	T_{WB1} (ms)	T_{WS1} (ms)	T_{PS} (ms)	T_{WS2} (ms)	T_{WB2} (ms)	T_{WB3} (ms)	T_{WS3} (ms)	T_{WB4} (ms)	T^*_{total} (ms)	T_{total} (ms)
Chrome	352 / 335	0 / 0	2 / 758	0 / 0	377 / 409	0 / 0	0 / 0	0 / 0	731 / 1502	754 / 1518
Firefox	451 / 466	0 / 0	2 / 602	0 / 0	514 / 537	0 / 0	0 / 0	0 / 0	967 / 1605	980 / 1621
IE	1208 / 1060	0 / 0	3 / 747	0 / 0	1271 / 1201	0 / 1	0 / 0	0 / 0	2482 / 3009	2494 / 3017
Microsoft Edge	351 / 342	0 / 0	3 / 559	0 / 0	407 / 397	0 / 0	0 / 0	0 / 0	761 / 1298	771 / 1309
Huawei Browser	362 / 362	0 / 0	3 / 701	0 / 0	408 / 419	0 / 0	0 / 0	1 / 0	774 / 1482	893 / 1592
Firefox for Android	310 / 296	0 / 0	2 / 549	0 / 0	346 / 401	0 / 0	0 / 0	0 / 0	658 / 1246	730 / 1338

"Busy-Hours" means reaching max throughput (70Kop/s) here
$T^*_{total} = T_{T1} - T_{T4}$, indicating total time without transmission delay

Table 16: Latency of SRP-2048 during Idle/Busy-Hours

	T_{WB1} (ms)	T_{WS1} (ms)	T_{PS} (ms)	T_{WS2} (ms)	T_{WB2} (ms)	T_{WB3} (ms)	T_{WS3} (ms)	T_{WB4} (ms)	T^*_{total} (ms)	T_{total} (ms)
Chrome	758 / 756	0 / 0	4 / 1552	0 / 0	994 / 914	0 / 2	0 / 0	0 / 0	1756 / 3224	1773 / 3241
Firefox	990 / 1016	0 / 0	3 / 1568	1 / 0	1210 / 1137	0 / 1	0 / 0	1 / 0	2205 / 3722	2296 / 3739
IE	2742 / 2400	0 / 0	3 / 982	0 / 0	3034 / 2678	0 / 1	0 / 0	0 / 0	5779 / 6061	5790 / 6071
Microsoft Edge	781 / 732	0 / 0	3 / 1486	1 / 0	823 / 822	0 / 6	0 / 0	0 / 0	1608 / 3046	1618 / 3055
Huawei Browser	798 / 779	0 / 0	4 / 1414	0 / 1	844 / 1004	2 / 1	0 / 0	0 / 0	1648 / 3199	1740 / 3405
Firefox for Android	684 / 726	1 / 1	3 / 1343	0 / 0	743 / 830	1 / 1	1 / 0	0 / 1	1433 / 2902	1521 / 3055

"Busy-Hours" means reaching max throughput (30Kop/s) here
$T^*_{total} = T_{T1} - T_{T4}$, indicating total time without transmission delay

Table 17: Latency of SPAKE2+ during Idle/Busy-Hours

	T_{WB1} (ms)	T_{WS1} (ms)	T_{PS} (ms)	T_{WS2} (ms)	T_{WB2} (ms)	T_{WB3} (ms)	T_{WS3} (ms)	T_{WB4} (ms)	T^*_{total} (ms)	T_{total} (ms)
Chrome	48 / 47	0 / 0	1 / 101	0 / 0	56 / 70	10 / 10	5 / 5	0 / 0	120 / 233	134 / 248
Firefox	64 / 66	0 / 0	1 / 108	0 / 0	81 / 85	15 / 15	5 / 5	0 / 0	166 / 279	181 / 296
IE	199 / 192	0 / 0	1 / 99	0 / 0	230 / 252	63 / 63	5 / 5	0 / 0	498 / 611	510 / 623
Microsoft Edge	57 / 46	0 / 0	1 / 100	0 / 0	74 / 68	9 / 10	5 / 5	1 / 0	146 / 229	154 / 240
Huawei Browser	51 / 84	0 / 0	1 / 113	0 / 0	76 / 106	11 / 12	5 / 5	1 / 0	145 / 320	191 / 385
Firefox for Android	53 / 57	0 / 0	1 / 100	0 / 0	65 / 108	14 / 15	5 / 5	0 / 0	138 / 285	192 / 331

"Busy-Hours" means reaching max throughput (1600Kop/s) here
$T^*_{total} = T_{T1} - T_{T4}$, indicating total time without transmission delay

Table 18: Latency of Hash-based Authentication

	Chrome	Firefox	IE	Microsoft Edge	Huawei Browser	Firefox for Android
Delay without T_T (ms)	61	61	62	61	61	61
Total Delay (ms)	73	74	68	70	80	85