AsyncGBP$^+$: Bridging SSL/TLS and Heterogeneous Computing Power With GPU-Based Providers

Yi Bian, Fangyu Zheng, Yuewu Wang, Lingguang Lei, Yuan Ma, Tian Zhou, Jiankuo Dong, Guang Fan, and Jiwu Jing, *Member, IEEE*

***Abstract*—The rapid evolution of GPUs has emerged as a promising solution for accelerating the worldwide used SSL/TLS, which faces performance bottlenecks due to its underlying heavy cryptographic computations. Nevertheless, substantial structural adjustments from the parallel mode of GPUs to the serial mode of the SSL/TLS stack are imperative, potentially constraining the practical deployment of GPUs. In this paper, we propose AsyncGBP$^+$, a three-level framework that facilitates the seamless conversion of cryptographic requests from synchronous to asynchronous mode. We conduct an in-depth analysis of the OpenSSL provider and cryptographic primitive features relevant to GPU implementations, aiming to fully exploit the potential of GPUs. Notably, AsyncGBP$^+$ supports three working settings (offline/online/hybrid), finely tailored for various public key cryptographic primitives, including traditional ones like X25519, Ed25519, ECDSA, and the quantum-safe CRYSTALS-Kyber. A comprehensive evaluation demonstrates that AsyncGBP$^+$ can efficiently achieve an improvement of up to 137.8× compared to the default OpenSSL provider (for X25519, Ed25519, ECDSA) and 113.30× compared to OpenSSL-compatible liboqs (for CRYSTALS-Kyber) in a single-process setting. Furthermore, AsyncGBP$^+$ surpasses the current fastest commercial-off-the-shelf OpenSSL-compatible TLS accelerator with a 5.3× to 7.0× performance improvement.**

***Index Terms*—TLS 1.3, post-quantum cryptography, heterogeneous computing, graphics processing unit.**

Received 26 January 2024; revised 18 June 2024; accepted 29 September 2024. Date of publication 10 October 2024; date of current version 20 January 2025. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB3103301, in part by the National Natural Science Foundation of China under Grant 61902392, and in part by the CCF-AFSG Research Fund under Grant CCF-AFSG RF20230206. An earlier version of this paper was presented Proceedings of the 52nd International Conference on Parallel Processing, 2023 [DOI: 10.1145/3605573.3605620]. Recommended for acceptance by T. Jones. *(Corresponding author: Fangyu Zheng.)*

Yi Bian is with the School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100043, China (e-mail: bianyi18@mails.ucas.ac.cn).

Fangyu Zheng, Yuewu Wang, and Jiwu Jing are with the School of Cryptology, University of Chinese Academy of Sciences, Beijing 100043, China (e-mail: zhengfangyu@ucas.ac.cn; wangyuewu@ucas.ac.cn; jwjing@ucas.ac.cn).

Lingguang Lei and Yuan Ma are with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100089, China (e-mail: leilingguang@iie.ac.cn; mayuan@iie.ac.cn).

Tian Zhou is with the School of Cyber Security, University of Science and Technology of China, Hefei 230026, China (e-mail: weekdayzt@mail.ustc.edu.cn).

Jiankuo Dong is with the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China (e-mail: djiankuo@njupt.edu.cn).

Guang Fan is with Ant Group, Hangzhou 310023, China (e-mail: fanguang.fg@antgroup.com).

Digital Object Identifier 10.1109/TC.2024.3477987

I. Introduction

Secure Socket Layer (SSL) and Transport Layer Security (TLS) [1] are cryptographic protocols designed to provide secure communication over the Internet, ensuring the confidentiality, integrity, and authenticity of data. SSL/TLS has become the de facto standard for securing Internet communications and is widely used in e-commerce, online banking, and other sensitive online transactions. According to the report [2], as of December 2023, 99.9% of surveyed websites support SSL/TLS, and 33.0% and 66.9% of them choose TLS 1.2 and TLS 1.3 as the best practice for SSL/TLS protocols, respectively.

In SSL/TLS, public key cryptography is employed for initial key exchange and authentication between the client and server, while symmetric key cryptography is used for data encryption and decryption after the secure connection has been established. For example, X25519 [3], ECDSA [4] and Ed25519 [5] are currently recommended in TLS 1.3 for key exchange and digital signature, respectively. Compared with symmetric key cryptographic primitives, public key ones which are built upon hard mathematical problems, are slower by 1 to 2 orders of magnitude. Today, employing SSL/TLS as a security measure for communication has become a "routine practice". However, due to the introduction of time-consuming cryptographic operations, it inevitably incurs additional performance losses.

A. *Offloading SSL/TLS to GPUs and Its Challenges*

In order to minimize the performance overhead imposed by cryptographic operations, it is a common practice in the industry to employ hardware accelerators to perform computationally intensive cryptographic tasks. Hardware cryptographic accelerators are generally based on platforms such as ASICs, GPUs, and FPGAs [6], [7], [8], [9], [10], [11], [12], [13], which offload cryptographic operations from CPUs to dedicated computing units, thereby improving performance and reducing the resource consumption, especially for large-scale cryptographic computing scenarios, such as data centers. GPU-based cryptographic accelerators can act as coprocessors of CPUs to

TABLE I
PARAMETER COMPARISON OF TRADITIONAL AND POST-QUANTUM CRYPTOGRAPHY

	Public Key (bytes)	Private Key (bytes)	Ciphertext (bytes)	Speed (ops)
X25519 [3]	32	32	32	24172.30
Kyber768 [15]	1184	2400	1088	11940.57/10195.76 *
Expansion Ratio	37.00×	75.00×	34.34×	0.49×/0.42×

∗Kyber768's encapsulation and decapsulation speed on the Intel Xeon Gold 5128R is separated by a slash.

handle cryptographic operations exclusively. Studies [7], [8], [9], [10], [11] have demonstrated that this solution can achieve high throughput with significant advantages.

However, previous studies in this area have mostly been confined to experimental or proof-of-concept levels [7], [11], [12], [13], [14], without practical deployment in real-world production environments. Many of these works concentrated on pure cryptographic algorithm implementations, without considering their real-world workhorse delivery [7], [11], [14], while others focused on cryptanalysis (e.g., brute-force attack) using GPUs [12], [13]. If the computing power of GPUs for cryptographic operations cannot be effectively integrated into applications, then any high performance achieved is merely an empty promise.

Unfortunately, applying GPU-based high-performance cryptographic algorithms to SSL/TLS implementations is not merely a matter of transitioning cryptographic algorithms from CPUs to GPUs. It actually represents a comprehensive system engineering challenge. One of the most complex challenges arises from the fundamental differences between CPU-based SSL/TLS protocol stacks (such as OpenSSL) and GPU architectures, posing difficulties in direct migration or usage. Additionally, unlike standard implementations of cryptographic primitives on CPUs, GPUs have different advantages and disadvantages when handling various computational workloads. Therefore, when integrating with SSL/TLS implementations, the operating modes of cryptographic primitives must be carefully considered to ensure collaboration with GPUs.

The challenges become even more severe when employing a quantum-safe setting. Post-quantum cryptography, sometimes referred to as quantum-safe cryptography, is one of the most promising approaches to defend against quantum computing attacks. Kyber, short for CRYSTALS-KYBER, is a lattice-based post-quantum key establishment mechanism (KEM) [15] that won the NIST post-quantum cryptography standardization and has been implemented in Chrome and Google Servers for SSL/TLS [16]. However, compared to traditional public key algorithms, Kyber involves more complex workflows, higher computational complexity, and a 30-fold size increase of key and ciphertext as shown in Table I. When using GPU acceleration, efficient implementation and optimizing memory access must be additionally considered.

B. Contributions

In this paper, we propose AsyncGBP$^+$ (**Async**hronous **G**PU-**B**ased **P**rovider + online/offline/hybrid settings and quantum-safe support), a framework that harnesses GPUs to accelerate cryptographic primitives while staying full compatibility with OpenSSL. This enables easy integration of this powerful cryptographic workhorse into existing systems. Specifically, we make the following contributions:

- Firstly, we conducted an in-depth analysis of the operational mechanism of the OpenSSL provider and explored the cryptographic primitive features of interest to GPU implementations. We propose an OpenSSL-compatible GPU-based cryptographic primitive acceleration framework named AsyncGBP$^+$. This framework enables efficient conversion between synchronous and asynchronous cryptographic operations,effective data aggregation, and reasonable GPU scheduling.
- Secondly, a three-level architecture for AsyncGBP$^+$ was designed, featuring three different working settings. We optimized both traditional cryptographic primitives (e.g., X25519, Ed25519 and ECDSA) and quantum-safe cryptographic primitives (e.g., Kyber). AsyncGBP$^+$ is tailored to the unique characteristics of various public key cryptographic primitives, ensuring full compatibility with the OpenSSL provider mechanism. It consistently delivers high performance in GPU implementations and can be seamlessly integrated into SSL/TLS implementations.
- Finally, a series of comprehensive experiments with the OpenSSL built-in utility show that AsyncGBP$^+$ can effectively bridge synchronous and asynchronous modes of cryptographic operations to fully leverage its SIMT feature. Empirical results demonstrate that AsyncGBP$^+$ can output up to 97% of GPU's local performance on RTX 3070, resulting in an improvement ranging from 32.77× to 137.81× compared to the default OpenSSL provider and oqs-provider in a single-process setting.

Furthermore, AsyncGBP$^+$ significantly outperforms the current fastest commercial-off-the-shelf OpenSSL-compatible TLS accelerator, achieving a 5.3× to 7.0× performance improvement. To the best of our knowledge, this is the first work that successfully bridges the gap between GPU-based cryptographic accelerators and the OpenSSL provider mechanism, resulting in significant performance improvements.

II. PRELIMINARY

This section introduces the TLS 1.3 protocol, the OpenSSL provider mechanism, and the basic theory of traditional and post-quantum cryptography. It provides a conceptual introduction only. A more detailed analysis and implementation of each mechanism will be presented in later sections.

A. OpenSSL Provider and ASYNC_JOB

As an open-source cryptographic library, OpenSSL is widely used in SSL/TLS implementations [17]. To transparently utilize third-party cryptographic implementations, OpenSSL has proposed both engine and provider mechanisms, considering the latter as a future focus [18]. A provider is a unit of code that provides one or more implementations of various operations for different cryptographic algorithms, either as a built-in or dynamically loaded module.

The analysis of the OpenSSL provider mechanism is a significant part of our work. We will introduce a detailed overview of the provider mechanism in Section III.

B. TLS 1.3 and Its Underlying Cryptographic Suites

Transport Layer Security (TLS) is a protocol that protects online communications. As the latest version, TLS 1.3 [19] enhances both the security and performance of the protocol, optimizes the handshake protocol, removes weak cryptographic algorithms, and adds advanced algorithms to provide better security, such as X25519 and EdDSA.

1) ECDSA, X25519 and EdDSA: Elliptic Curve Cryptography (ECC) relies on the elliptic curve discrete logarithm problem (ECDLP), i.e., finding the discrete logarithm (d) of a random elliptic curve element ($Q = [d]P$) given a publicly known base point P is infeasible. For most ECC schemes, the core operation is *point multiplication*, i.e., computing a multiple of a point $Q = [d]P$.

ECDSA. The Elliptic Curve Digital Signature Algorithm (ECDSA) is a widely used public-key cryptographic algorithm for digital signatures. The elliptic curve it employs is the Weierstrass curve, which is defined over a prime field as equation (1), where $a, b \in \mathbb{F}_p$, and $4a^3 + 27b^2 \neq 0 \pmod{p}$. Different curve parameters determine distinct elliptic curves, with common choices including curves defined by NIST standards [20], such as P256 or P384.

$$y^2 \equiv x^3 + ax + b \tag{1}$$

In this paper, we focus on ECDSA-P256.

X25519. Let $\mathbb{F}_p$ be a prime field of odd characteristics, where $p = 2^{255} - 19$; a Montgomery curve over $\mathbb{F}_p$ is defined as

$$E_{a,b}/\mathbb{F}_p : by^2 = x^3 + ax^2 + x. \tag{2}$$

where $a, b \in \mathbb{F}_p$, $a \neq \pm 2$.

Taking a scalar d and an x-coordinate of point P as input and producing an x-coordinate of $[d]P$ as output, a Montgomery ladder is an efficient x-coordinate-only algorithm to calculate point multiplications. Combined with the Diffie-Hellman key exchange protocol, Montgomery curves were adopted for establishing secure communications over the Internet [3], known as the X25519 key agreement protocol.

EdDSA. Montgomery curves have birationally equivalent Edwards versions which support the fast complete formulas for the elliptic-curve group operations. Let $\mathbb{F}_p$ be a prime field of odd characteristics; a twisted Edwards curve over $\mathbb{F}_p$ is defined as

$$E_{c,d}/\mathbb{F}_p : \alpha x^2 + y^2 = 1 + \beta x^2 y^2. \tag{3}$$

where $\alpha\beta(\alpha - \beta) \neq 0$.

Using a variant of the Schnorr signature based on twisted Edwards curves, EdDSA [5] is a high-performance digital signature scheme with small signatures and public keys, which has gradually replaced ECDSA in relevant applications. Using the same prime $p = 2^{255} - 19$ as X25519, the most widely used instance of EdDSA is called Ed25519, which we will implement in this paper.

2) CRYSTALS-Kyber: CRYSTALS-Kyber is an IND-CCA2-secure key encapsulation mechanism (KEM), whose security is based on the hardness of solving the Module-LWE problem [21]. Compared to authentication, countering the threat of "Harvest Now, Decrypt Later" (HNDL) attacks is more urgent. This is also why the Chromium project prioritizes the quantum transition for KEM [16].

Kyber provides three different parameter sets (Kyber512/768/1024), each corresponding to different levels of security strength (AES-128/192/256). The official recommended parameter set is Kyber768, which can achieve over 128 bits of security against all known classical and quantum attacks. Kyber is constructed in two stages: the first stage builds Kyber.CPAPKE, then a slightly tweaked Fujisaki-Okamoto transform [22] is used to construct the IND-CCA2-secure KEM, referred to as Kyber.CCAKEM. To obtain more information, please refer to the Ref [23].

III. Design

In this section, we present the technical challenges, overall architecture, and working settings of AsyncGBP$^+$.

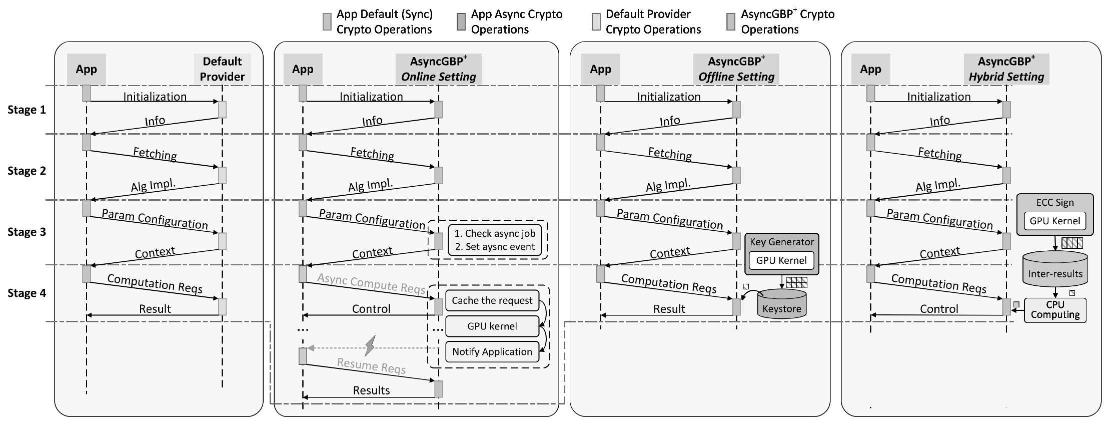
A. Technical Challenges

Our primary goal is to deploy GPUs in practical applications, reducing migration costs while maximizing their performance. Although GPUs have inherent advantages in computing power, achieving optimal performance of the overall system presents many technical challenges.

Architectural Adjustments. There are significant differences in the working settings of GPUs and OpenSSL, especially in calling methods and data processing capabilities. Although OpenSSL provides an asynchronous mode to offload cryptographic tasks, it resembles an interface from the developer's perspective and cannot seamlessly integrate with the parallel mode of GPUs. To achieve seamless integration, the architecture must be meticulously designed to bridge the gap between GPU and OpenSSL.

The Balance between User Experience and Performance. To ensure the implementation of underlying cryptographic primitives is transparent to upper-level applications, a balance must be struck between usability and performance. Previous studies often employ customized interfaces to demonstrate their peak performance, which is not feasible for real-world applications. Therefore, adopting a standard interface is necessary to ensure ease of use while minimizing performance degradation.

Differences in Data Dependency of Cryptographic Primitives. The data dependency of cryptographic primitives does not affect CPU computing but significantly impacts GPU computing. It determines the request mode (synchronous/asynchronous), the kernel's batch size, and the potential overlap of kernel execution and data transfer, ultimately influencing throughput. Therefore, we need to thoroughly analyze the data dependency of cryptographic primitives and provide appropriate solutions for them.



(a) Default Provider. (b) AsyncGBP$^+$ Online Setting. (c) AsyncGBP$^+$ Offline Setting. (d) AsyncGBP$^+$ Hybrid Setting.

Fig. 1. The Workflow of Default Provider and AsyncGBP$^+$. (*Stage 1 involves provider loading and initialization, stage 2 focuses on algorithm implementation fetching, stage 3 is algorithm parameters configuration and stage 4 is cryptographic computation.*)

B. Workload Analysis

To overcome the above challenges, we analyze the cryptographic workload of TLS from two aspects: 1) analyzing the OpenSSL default provider workflow to find the optimal combination of GPU and provider mechanism; 2) evaluating data dependencies of different cryptographic primitives to identify the most efficient computation settings.

1) Analysis of OpenSSL Provider: Although OpenSSL provider can offer many implementations of cryptographic primitives, its primary workflow can be summarized into four general stages, as shown in Fig. 1(a).

1) *Stage 1: Provider Loading and Initialization.* In this stage, an application explicitly or implicitly invokes the provider initialization API. OpenSSL's component `Core` loads the provider into the memory, calls the entry point for initialization, and obtains its basic information, including the name, properties, supported algorithm implementations, and more.
2) *Stage 2: Algorithm Implementation Fetching*. Before utilizing an algorithm, the application first needs to query and obtain the target implementation. The specific process varies depending on whether the implementation has been cached. Firstly, the cache is searched. If the first search fails, `Core` initiates a query to the provider and then caches the result.
3) *Stage 3: Algorithm Parameters Configuration.* Algorithm parameters related to cryptographic computations, such as algorithm types, operation types, and keys, are set during this stage.
4) *Stage 4: Cryptographic Computation.* The application initiates cryptographic computation requests using high-level interfaces (i.e., EVP APIs) and obtains results, which represents the real computing process and reflects the performance of the provider.

Any provider requires the first two stages of the workflow, which are preparation processes that need to be executed only once. The third stage is responsible for initializing the parameters of the cryptographic task. Whether parameters need to be reconfigured for each cryptographic computation depends on specific application requirements. As these processes are not computationally intensive, the CPU is the appropriate processor for executing them.

Stages 3 and 4 are directly associated with cryptographic operations, so their design significantly affects the system's overall performance. The default provider employs cryptographic requests in synchronous mode, as shown in Fig. 1(a), which means that the application is blocked while waiting for the result, potentially resulting in performance degradation in large-scale request scenarios. Furthermore, for synchronous cryptographic requests, only one cryptographic operation can be processed at a time, which presents a huge gap with the batch pattern of GPUs.

Fortunately, OpenSSL also supports an asynchronous mode, implemented by the fiber-based `ASYNC_JOB` [24], which provides the infrastructure to offload time-consuming cryptographic operations to hardware accelerators. The function `ASYNC_start_job` launches an asynchronous job, while `ASYNC_pause_job` is called to return control to the caller after the cryptographic computation requests have been delivered to the accelerator. When receiving the notification of completed computation, the application resumes execution by calling `ASYNC_start_job` again with the previously suspended `ASYNC_JOB` as a parameter and subsequently retrieves the results. This lays the groundwork for implementing an OpenSSL provider tailored for GPUs, marking the inception of our project.

2) Analysis of SSL/TLS Cryptographic Primitives: Unlike typical implementations based on CPUs, GPU-based implementations of cryptographic primitives need to consider the internal structure of algorithms when integrating with higher-level applications. Hence, we conducted a comprehensive analysis of cryptographic primitives, classifying them into three types based on their data dependencies.

Fully data-dependent operations. These refer to cryptographic operations that must wait for external data before computation. Most cryptographic operations fall into this type, such

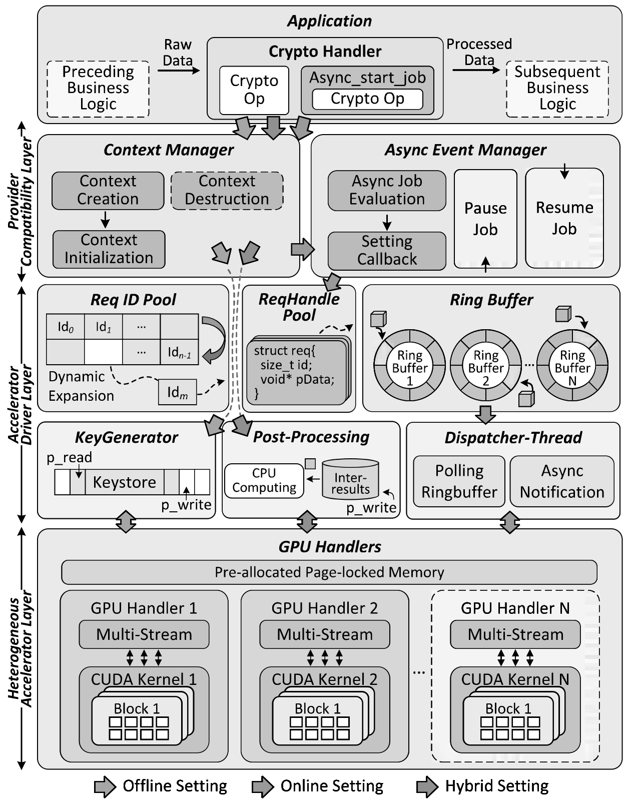


Fig. 2. The Overall Architecture of AsyncGBP$^+$. (*The dark gray arrows represent the offline setting workflow, the green diagonal arrows indicate the online setting workflow and the blue diagonal arrows denote the workflow of the hybrid setting. The yellow rounded rectangles denote threads, while the other rounded rectangles present components.*)

as Kyber encapsulation/decapsulation, EdDSA signature generation/verification, X25519 key exchange, and ECDSA signature verification, etc. This feature poses challenges in implementing high-performance cryptographic operations for requests in synchronous mode. Hence, we propose an online computing setting to address this issue, as shown in Fig. 1(b). We convert cryptographic requests from synchronous to asynchronous mode, aggregate them using a lightweight request caching mechanism, and utilize GPU-based cryptographic primitives to achieve high-performance cryptographic computations.

Zero data-dependent operations. These denote cryptographic operations that can be computed independently without waiting for any request data. Key generation is a typical example, which refers to the process of generating keys using either a true random number generator (TRNG), pseudorandom number generator (PRNG), or cryptographic primitives. For example, public-key cryptography takes a random number as input and generates private and public keys through specific cryptographic primitives. This allows us to accelerate key generation using pre-computation techniques, as illustrated in the offline setting of Fig. 1(c). Specifically, we established a "single producer-single consumer" model, the producer runs in the background as an independent thread ("KeyGenerator" thread as shown in Fig. 2), continuously generating keys using GPU acceleration and storing them in the keystore, while the

Algorithm 1 ECDSA Signature Generation

```
Input: Private key $d$, digest $e$
Output: Signature $(r, s)$
1: $k \in_R \mathbb{Z}_n^*$    ▷ Offline pre-computation
2: $(x_1, y_1) = kG$    ▷ Offline pre-computation
3: $r = x_1$ mod $n$. If $r = 0$ then go to step 1    ▷ Offline pre-computation
4: $s = k^{-1}(e + dr)$ mod $n$. If $s = 0$ then go to step 1    ▷ Online computation
5: return $(r, s)$
```

TABLE II
COMPARISON OF FEATURES IN DIFFERENT SETTINGS OF ASYNCGBP$^+$

	Data Dependency	Request Type	Execution Unit	Cryptographic Operations
Offline Setting	○	Sync	GPU	Key Generation
Online Setting	●	Async	GPU	X25519, ECDSA_Verify Ed25519 Sign/Verify Kyber Encaps/Decaps
Hybrid Setting	◐	Sync	CPU+GPU	ECDSA-Sign

consumer retrieves keys from the keystore as required. This technology can effectively reduce the latency associated with key generation requests. Additionally, it decouples the GPU's batch size from the number of requests, regardless of the request mode (synchronous or asynchronous). We chose synchronous requests to minimize system latency.

Partial data-dependent operations. These refer to cryptographic algorithms that involve both zero and fully data-dependent operations, such as ECDSA signature generation. According to Algorithm 1, steps 1-3 of ECDSA signature generation and the inverse of k in step 4 are independent of external requests. Therefore, we decompose the algorithm into zero data-dependent operations (steps 1 to 3, and k^{-1}) and a fully data-dependent operation (step 4), and propose a hybrid computing setting as presented in Fig. 1(d). Scalar multiplication is a complex computation in elliptic curve cryptography, so we delegate this task to the GPU, which requires cryptographic requests in asynchronous mode. The intermediate results, r and the calculation of k^{-1}, are pre-computed by the GPU and stored in the buffer, and the CPU utilizes them to compute the final signature.

We summarize the above analysis in Table II. Later, we will introduce the design of three different working settings, namely online, offline, and hybrid settings.

C. Overall Architecture and Three Working Settings of AsyncGBP$^+$

Based on the analysis of TLS cryptographic workloads, we propose AsyncGBP$^+$, a compact and high-performance TLS acceleration framework, as shown in Fig. 2. AsyncGBP$^+$ comprises three layers: the Provider Compatibility Layer (*ProvComp*), the Accelerator Driver Layer (*AccDrv*), and the Heterogeneous Accelerator Layer (*HeteAcc*).

- The first layer, *ProvComp*, ensures compatibility with the OpenSSL provider mechanism. As mentioned earlier, asynchronous requests are necessary to speed up non-zero data-dependent cryptographic operations. Hence, in addition to the context manager, we also introduce an asynchronous event manager to support such requests.

- *AccDrv* serves to cache requests and schedule computing tasks, which impacts the performance of the solution. We adopt a compact design that allows sharing of components between different working settings. To fully harness GPU computing power, we design a lightweight request caching scheme, which includes a request ID pool, request handle pools, and ring buffers. By building the chain relationship of $\langle$*ID, ReqHandle, Request*$\rangle$, we achieve a lightweight and efficient request aggregation.
- The third layer is *HeteAcc*, which provides a high-performance implementation of cryptographic primitives. It integrates optimized GPU-based cryptographic primitives and utilizes methods such as pre-allocation, page-locked memory, and CUDA streams to improve performance.

In terms of working settings, AsyncGBP$^+$ supports online, offline and hybrid settings, which adopt a compact design, allowing different settings to share components.

Online Setting. This setting is one of the main working settings supported by AsyncGBP$^+$, particularly applicable for fully data-dependent cryptographic operations. Cryptographic requests initiated asynchronously are processed by the lightweight request caching scheme and stored in the corresponding ring buffer. The dispatcher thread is responsible for polling ring buffers to find pending requests. Once a request is detected, the dispatcher will call the GPU handler to perform computations in batches, and notify the upper-level application to obtain the computation results.

Offline Setting. This setting is suitable for zero data-dependent cryptographic operations, such as key generation. Cryptographic computation requests are synchronous in this setting. To mitigate the latency introduced by real-time computation, we design a low-latency key generation scheme using the pre-computation technique. This scheme can be summarized as a "single producer-single consumer" model: a named "KeyGenerator" thread runs continuously in the background, invoking the GPU-based handler to generate keys and filling the pre-allocated buffer named "keystore". The keys are directly fetched from the corresponding keystore based on the type of cryptographic requests.

Hybrid Setting. This setting is a fusion of the first two settings, which is suitable for partial data-dependent cryptographic operations such as ECDSA signature generation. The cryptographic requests in this setting are asynchronous, so the lightweight request caching scheme is also utilized. Unlike the online setting, computation in the hybrid setting is performed collaboratively by both the CPU and GPU, rather than by the GPU alone. This setting deconstructs operations based on data dependencies. Operations with zero data dependency are processed by the GPU in the offline setting and the results are stored in an intermediate result ("inter-result") pool. Fully data-dependent operations are processed in the online setting, where the CPU obtains the required intermediate result from the "inter-result" pool and then computes the final result.

IV. AsyncGBP$^+$ Implementation

In this section, we will introduce the specific implementation of each layer from top to bottom as illustrated in Fig. 2.

A. Provider Compatibility Layer

One of the design goals of AsyncGBP$^+$ is to maintain compatibility, including support for OpenSSL context and the provider mechanism. In addition, a mechanism is required in asynchronous mode to notify the caller when the computation results become available. Therefore, we introduce two components in this layer to address these issues.

1) Context Manager: The Context Manager (CM) is responsible for the life-cycle management of the context. Specifically, in the algorithm initialization stage, CM creates a context and binds the application-defined parameters to it. In the cryptographic computation stage, CM associates the computation results to the context for retrieval by the application.

Due to the lack of support for post-quantum cryptography in the OpenSSL default provider, it is necessary for us to create the relevant context to invoke PQC through OpenSSL. Following the design principles of the OpenSSL provider [25], [26] and taking inspiration from the implementations of OpenSSL's ECX and RSA [17], we designed and implemented the PQC context, including key management and key encapsulation. Specifically, we built a structure named `PQC_KEY`, which includes key data such as OpenSSL library context, key type, pointers to public and private keys, and key length. Then, we referred to the relevant OpenSSL code to implement various functions of key management and key encapsulation for post-quantum cryptographic primitives.

2) Asynchronous Event Manager: The Asynchronous Event Manager (AEM) is specifically designed to handle asynchronous events and serves as a critical component of the OpenSSL asynchronous model. The functionality of the AEM is triggered exclusively upon the invocation of the EVP APIs that participate in the actual computation. Specifically, AEM ascertains whether the current job is asynchronous. If the current job is identified as asynchronous, AEM registers an asynchronous notification handle for the job. Once the cryptographic request is successfully cached, AEM suspends the asynchronous job and returns control to the caller.

Typically, the notification handle can be either a file descriptor or a callback. Read and write operations on file descriptors rely on system calls, leading to frequent switching between kernel and user mode. In high-concurrency environments, this pattern can result in significant performance overhead. In contrast, callbacks, based on the event-driven model, are inherently suitable for asynchronous architectures, requiring execution only in user mode. As a result, we employ the callback as the notification handle.

Additionally, we integrate some necessary functions into this layer to be compatible with the OpenSSL provider mechanism, such as provider initialization and teardown, cryptographic algorithm implementation queries. Considering that the HKDF [27] algorithm for TLS 1.3 cipher suite is not yet implemented by AsyncGBP$^+$, we adopt a hybrid strategy to maximize the availability of the framework. Specifically, for cryptographic algorithms supported by the AsyncGBP$^+$, the implementations are set to high priority and provided to the OpenSSL `libcrypto`. Otherwise, we mark them as low priority and reroute cryptographic requests to the default implementations of OpenSSL.

B. Accelerator Driver Layer

In AsyncGBP$^+$, GPUs serve as the final handlers of cryptographic operations, showcasing significant differences in working mode compared to OpenSSL. Therefore, we adopt the following measures to bridge the gap between OpenSSL and GPU working mode.

1) Lightweight Request Caching: Copying the data required for cryptographic operations directly to the ring buffer for collecting requests is inappropriate, as it causes redundant data copying, reducing the efficiency of request collection. Therefore, we design a lightweight request caching scheme. In this approach, each cryptographic request is assigned a unique identifier. This identifier is utilized to retrieve a corresponding request instance and associate itself with the data associated with cryptographic computation. In this manner, we establish a chain relationship of $\langle$*ID, ReqHandle, Request*$\rangle$. By collecting identifiers, we achieve efficient and lightweight request aggregation, while eliminating redundant data copying.

To mitigate the overhead caused by temporary memory allocation, we establish an ID pool and a request handle pool, both of which support dynamic expansion as needed. The identifiers are stored in different ring buffers based on the type of cryptographic requests, leveraging a space-time trade-off strategy to reduce the cost of sorting different types of requests. In addition, we employ a "single producer, single consumer" model and a lock-free design to prevent data race and minimize the performance overhead incurred by accessing the ring buffer. After multiple experiments, we configure the initial sizes of the ID pool and the request handle pool as 8192, with the ring buffer size as 4096.

2) Request Dispatcher: The request dispatcher periodically retrieves pending cryptographic requests from ring buffers and invokes the corresponding GPU handlers based on their type. Thanks to the design of a dedicated ring buffer for each type of cryptographic operation, we no longer need to spend time sorting cryptographic requests.

In the multi-threaded system, concurrently accessing the same ring buffer by different threads causes conflicts and unnecessary waiting. To solve the issue, each dispatcher is assigned an initial offset that enables it to access different ring buffers at the same time, which is a small trick but significantly reduces latency.

After calling the GPU handler to perform cryptographic computations, the dispatcher also needs to notify the application that the computation results are ready through the asynchronous notification handle. Due to the chain relationship of $\langle$*ID, ReqHandle, Request*$\rangle$, the computation results are already in the memory allocated by the application, so the dispatcher can directly recycle the ID and request handle instance for the next use. Excessive dispatchers not only consume CPU resources but also intensify the data race of ring buffers, resulting in inefficient GPU utilization. After experiments, we determine that the optimal number of dispatchers is 4.

3) Efficient Offline Key Generation: Key generation is independent of external requests and can be precomputed. Combining these two features, we propose an efficient offline key generation scheme. It is built on the "single-producer-single-consumer" model, where both parties read and write keys within a shared buffer. In particular, we construct a lock-free buffer named "keystore" based on an array to store generated keys. It features two atomic read and write pointers, allowing access to data without locking.

The design of the "consumer" is relatively straightforward, involving the simple task of checking for a sufficient number of keys within the "keystore" and fetching them as needed. The producer is built as a standalone thread named "KeyGenerator", which continuously generates keys in the background using GPU acceleration. To enhance the speed of key fetching, the "keystore" is designed to be significantly larger than the GPU batch size, inevitably requiring data copying. Consequently, the batch size and "keystore" size are closely related to performance. We conduct a series of experiments to identify optimal configurations in Section VI. Based on practical evaluation, we set the batch size to 11776 instances and the "keystore" size to 11776×8 instances. Taking Kyber768 as an example, each instance represents a Kyber768 key. Considering that its private key structure includes the public key, it can store only the private key, with a size of 2400 bytes. Therefore, the size of the keystore is $11776 \times 8 \times 2400$ bytes ≈ 215.6 MB.

4) Two-Step ECDSA Signature Generation: Based on the analysis in Section III, we decompose the ECDSA signature generation process into two primary parts: offline computation and online computation. The former comprises the first three steps of Algorithm 1, while the latter executes the fourth step of Algorithm 1. Within the offline computation stage, we configure it as an independent process running in the background. The GPU handler is tasked with performing ECC scalar multiplication, modulo reduction of the x-coordinate, and storing intermediate results (r and k^{-1}) in a lock-free buffer based on arrays. The required random numbers are generated in batches by the CPU and transferred to the GPU. As there are no data dependencies in these operations, they can be performed in the background with arbitrary batch size. After actual evaluation, we set the batch size to 2048, and the intermediate result buffer size is four times the batch size. The size of each intermediate result is 64 bytes, so the intermediate result buffer size is $2048 \times 4 \times 64$ bytes ≈ 512 KB.

A complete ECDSA signature generation also requires the private key and digest to participate in the calculation, which must be performed in online setting due to their full data dependencies. The operations required in this step include one modular addition and two modular multiplications. As these operations are not complicated, we choose to use the CPU to perform computations to reduce latency.

C. Heterogeneous Accelerator Layer

The Heterogeneous Accelerator Layer focuses primarily on implementing high-performance cryptographic algorithms. The collected cryptographic requests are forwarded to this layer for computation under the control of the dispatcher. We provide CUDA-based handlers for the cryptographic primitives required by TLS 1.3, including Kyber768 keyGen/Encaps/Decaps, X25519 key exchange, and signature generation and verification for both Ed25519 and ECDSA.

Different working settings are configured with different kernel scheduling strategies. In the offline and hybrid settings, the GPU kernels continuously run in the background with a predetermined batch size to perform cryptographic computations. In the online setting, we implement request-driven kernels that activate specific GPU kernels based on request type and configure kernel execution parameters according to the number of cryptographic requests.

Since we utilize discrete GPUs as heterogeneous accelerators, data transfer between host and device is inevitable and may adversely influence the performance of the GPU. As a solution, we leverage the asynchronous transfer mechanism and page-locked memory provided by CUDA to improve the transfer efficiency between the host and the device. It is worth noting that allocating too much page-locked memory can affect the performance of the CPU. Therefore, it is necessary to reasonably set the data size that can be processed simultaneously. According to repeated trials, each GPU handler is assigned a stream capable of transmitting up to 2048 requests. The number of streams is 4, which is equal to the number of dispatchers.

1) Tensor Core-Based Kyber: For the implementation of Kyber.CPAPKE, we follow the optimization method proposed by Wan et al. [10], and propose an optimization scheme for the implementation of Kyber.CCAKEM.

Kyber exploits a customized NTT in its algorithms, such as Equation (4) and (5). Only $n/2$ coefficients of a vector are involved in an NTT result. The NTT of a polynomial $f \in R_q$ is a vector of 128 degree-1 polynomials and can be represented as

$$NTT(f) = \hat{f} = (\hat{f}_0 + \hat{f}_1 X, \hat{f}_2 + \hat{f}_3 X, \cdots, \hat{f}_{254} + \hat{f}_{255} X)$$

with

$$\hat{f}_{2i} = \sum_{j=0}^{127} f_{2j} \zeta^{(2\boldsymbol{br}_7(i)+1)j} \quad (4)$$

$$\hat{f}_{2i+1} = \sum_{j=0}^{127} f_{2j+1} \zeta^{(2\boldsymbol{br}_7(i)+1)j} \quad (5)$$

where twiddle factor ζ is the 256-th root of unity, $\boldsymbol{br_7}$ represents a 7-bit bit-reverse function.

Based on the above observations, Wan et al. adopted a straightforward routine combined with techniques such as precomputation [10], as shown in Fig. 3. The essence of Wan et al.'s approach lies in transforming the Number Theoretic Transform (NTT) operation into a "large" matrix multiplication of size 128 × 128 and processing this large matrix using the native "16 × 16" matrix multiplication instructions of Tensor Core [28]. This strategy has delivered highly favorable outcomes, primarily because the dimension $n = 128$ is relatively small.

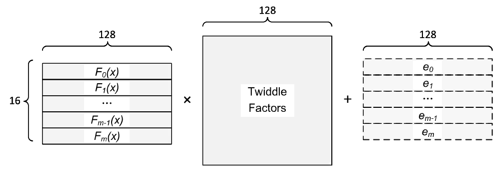


Fig. 3. Tensor Core-based NTT. (*The blue diagonal dashed box on the left describes the polynomial vector, while the orange one on the right represents the optional polynomial vector.*)

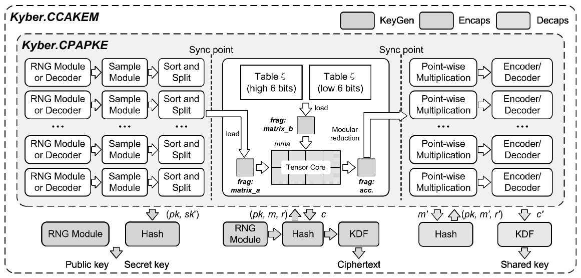


Fig. 4. General overview of implemented Kyber.

We implement Kyber as is shown in Fig. 4. In our implementation, each thread processes one Kyber instance and all threads operate in SIMT mode. The optimized implementation of Kyber can be divided into two parts, Kyber.CPAPKE and Kyber.CCAKEM. Firstly, we further optimize the previous work [10] by using the PTX instructions to improve memory access for Kyber.CPAPKE at the instruction level. Secondly, we refactor the GPU implementation and implement a complete Kyber.CCAKEM. To be more specific, we replaced basic data types with vector types and employed vector `load/store` instructions to achieve wide data loading and storing.

2) Two Working Setting of ECC Implementations: Based on the analysis of data dependencies presented in Section III, it is determined that X25519 key exchange, Ed25519 signature generation and verification, as well as the ECDSA signature verification can only be performed in online setting. On the other hand, the ECDSA signature generation can be executed in hybrid setting, including both online and offline computations. In the following discussion, we present our approach to optimizing the implementation of ECC primitives from the perspective of computing settings.

Offline and Online Computing Techniques. The implementation of Elliptic Curve Cryptography can be divided into three levels: finite field arithmetic, point arithmetic, and scalar multiplication. In this study, we follow the method proposed in Ref [8] and [29].

For finite field arithmetic, we optimize large integer addition/subtraction, multiplication/multiplication-addition, and

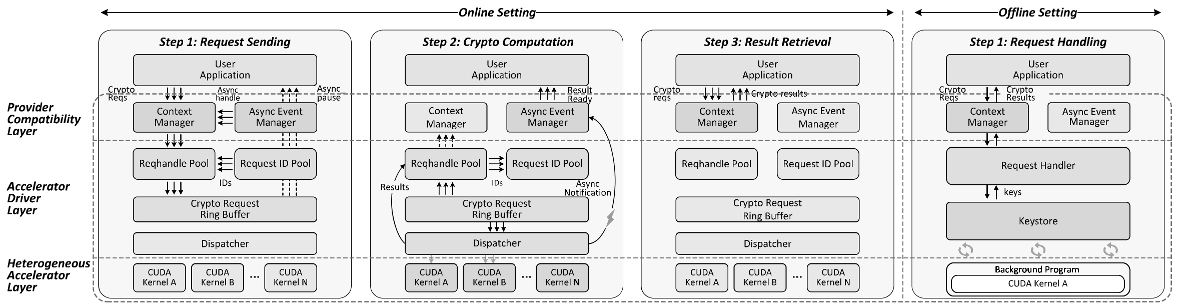


Fig. 5. An overview of the Cryptographic Computation Stage (Stage 4) in AsyncGBP$^+$ Online and Offline Settings. (*The colored rectangles show the active modules, while gray ones indicate inactive modules at this step. The rotating arrow shows the program is continuously filling the keystore.*)

square operations by using PTX instructions, and use a method that only requires one round of carry-reduce.

The main operation of X25519 is "x-coordinate only" unknown point scalar multiplication. We utilize the standard Montgomery ladder method to implement X25519 scalar multiplication and minimize the use of temporary variables to fully utilize register resources.

For ECDSA and Ed25519, there are similarities in their optimization methods. In general, they both utilize pre-computation techniques to speed up scalar multiplication. For fixed-point scalar multiplication, a portion-by-portion addition method [29] is used to generate an offline pre-computation table. However, for unknown-point scalar multiplication, the method of generating the pre-computation table offline is no longer suitable as the point is undefined. A fixed-window approach [30] is employed to generate the pre-computation table online.

The difference between Ed25519 and ECDSA lies in the coordinate system and the representation of points. Ed25519 employs the extended twisted Edwards coordinate system, with its pre-computation table in the form of $(x, y, x \times y, y - x, y + x)$, while ECDSA uses the affine coordinate system with a pre-computation table structured as (x, y). In addition, the size of the pre-computation table also varies. Assuming that each point consists of l coordinates, the size of the pre-computation table is $l \times 256 \times n \times 2^m$ bits. We adopt the parameter recommended in Ref [8] and [29], i.e., $m = n = 16$, and the number of coordinates (l) of each point is 5 and 2 for Ed25519 and ECDSA, respectively. Consequently, their offline pre-computation table sizes are 160 MB and 64 MB respectively.

Hybrid ECDSA. For the hybrid implementation of ECDSA signature generation, we perform step 4 of Algorithm 1 on the CPU, i.e, $s = k^{-1}(e + dr) \bmod n$, where the k^{-1} and r are computed by the GPU and stored in intermediate result buffer. This step involves two modular multiplications and one modular addition, all of which are implemented using inline assembly language.

V. Resulting Workflow and Deployment of AsyncGBP$^+$

In this section, we present the workflow of three settings of AsyncGBP$^+$ in the cryptographic computation stage, and provide a detailed guideline on its deployment.

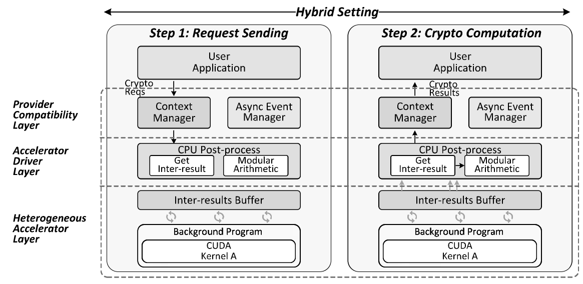


Fig. 6. An overview of the Cryptographic Computation Stage (Stage 4) in AsyncGBP$^+$ Hybrid Setting. (*The colored rectangles represent active modules, and gray ones indicate inactive modules during this step. The rotating arrow indicates that the program is running continuously in the background, filling the inter-result buffer in real time.*)

A. Workflow of AsyncGBP$^+$

Figs. 5 and 6 illustrate the workflow of the cryptographic computation stage of AsyncGBP$^+$ in its online, offline, and hybrid settings. The colored rectangles represent active modules, while gray rectangles indicate inactive modules during this step. The asynchronous nature of the framework facilitates the concurrent execution of cryptographic requests. Consequently, the actual execution process may not strictly adhere to that depicted in the figure.

The online setting comprises three steps: 1) Request Sending, 2) Cryptographic Computation, and 3) Result Retrieval, while the hybrid setting contains the first two steps, the offline setting encompasses only the Request Handling step. To improve the clarity of writing, we present the workflow of AsyncGBP$^+$ primarily focusing on the "online setting", and explain the working process of the other two settings in the form of notes.

Step 1: Request Sending or Request Handling. In this step, the application initiates a cryptographic request using the `ASYNC_start_job` API, which encapsulates the request as an asynchronous job and delivers it to the *ProvComp* layer. The incoming parameters are bound to a context by the Context Manager. Then, the Asynchronous Event Manager creates an asynchronous handle to notify the application when the computation is finished. Then, the context is delivered to the *AccDrv* layer.

AccDrv retrieves an available instance from the *ReqHandle Pool* based on the request ID. Once an available instance is found, *AccDrv* sets its status to in-use and associates it with the corresponding context. Lastly, the request ID is added to the corresponding ring buffer based on the cryptographic computation request type.

Notes on hybrid setting. Requests are synchronously processed in this setting and received by CPU-based post-processing thread for subsequent computation.

Notes on offline setting. Requests are synchronous in this setting, and the GPU-based key generator runs continuously as a background process, storing the generated keys in a buffer (keystore). When *ProvComp* receives a key generation request, the request handler fetches the required key from the keystore and returns it directly to the application. In low-volume request scenarios, this setting can achieve nearly zero-delay key generation.

Step 2: Crypto Computation. For online computing setting, *AccDrv* contains a crucial component for cryptographic computation, namely the dispatcher. It is responsible for primary tasks: scheduling GPU processing functions to perform cryptographic computation and notifying the caller through an asynchronous notification handle. Specifically, the dispatcher periodically checks ring buffers to identify any requests requiring processing. If there are no pending requests, the dispatcher enters a sleep state and waits for the subsequent polling cycle. Conversely, if any pending requests exist, the dispatcher fetches them and invokes the GPU handlers based on the primitive type.

Since the ring buffer only stores request IDs, the dispatcher needs to obtain the cryptographic requests through the triad ⟨*ID, ReqHandle, Request*⟩. The actual cryptography operations take place in the Heterogeneous Accelerator Layer. In the online setting, the dispatcher invokes corresponding GPU handlers for cryptographic computations based on the request type and binds the results to the corresponding *ReqHandles* through the triad.

As the computation process is asynchronous, the application remains unaware of the completion time for cryptographic computations. Therefore, it is essential for the dispatcher to notify the application to retrieve the computation result through an AEM-registered asynchronous handle. After the computation is finished, the request IDs and *ReqHandles* are reinitialized for subsequent use.

Notes on hybrid setting. The post-processing thread can be regarded as a special case of the dispatcher thread in this setting. It is specifically designed to handle ECDSA signature generation without polling the ring buffers. As mentioned earlier, operations such as scalar multiplication of ECDSA signature generation (step 1-3 of Algorithm 1) are performed in the offline setting, and results are stored directly in the intermediate result buffer. Therefore, the post-processing thread can directly retrieve these results and perform subsequent computations (step 4 of Algorithm 1) on this basis, and finally obtain the ECDSA signature result. As the request is synchronous, the upper-layer application can get the signature result directly without initiating the request again as in the online setting.

Step 3: Result Retrieval. When notified by an asynchronous handle, the application passes the previously paused asynchronous task as a parameter to the `ASYNC_start_job` function to resume it. The application directly jumps to the pause point, where it restores the context and continues execution. Since we employ a lightweight caching technology that eliminates redundant memory copies, the computation results are already stored in the memory allocated by the application. The *ProvComp* can return the results directly, thereby ending the entire cryptographic computation process.

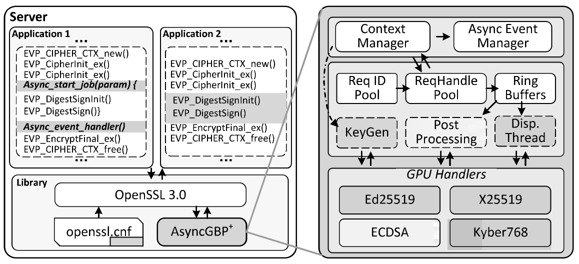


Fig. 7. Deployment to Applications. (*The left side illustrates the application with synchronous (orange area) and asynchronous (blue area) cryptographic computation requests. On the right, the simplified AsyncGBP$^+$ showcases online (blue area), hybrid (blue diagonal area), and offline (orange area) settings.*)

B. Deployment to Applications

The architecture of an application integrated with AsyncGBP$^+$ is illustrated in Fig. 7. The deployment includes three aspects.

Hardware Environment. To meet the hardware prerequisites for implementing AsyncGBP$^+$, application vendors are required to equip their servers with GPUs. If Kyber is required, the GPU must support the Tensor Core technique.

Software Environment. On the software side, the AsyncGBP$^+$ is provided in the shared object (.so) format. To utilize AsyncGBP$^+$ to accelerate applications, developers need to specify the path, name, and activation status in `opessl.cnf` configure file. Furthermore, OpenSSL 3.0 or higher is a prerequisite to leverage the provider mechanism. Finally, it is necessary to install the GPU driver and CUDA toolkit to enable GPU universal computing.

Application Adaptation. For offline computing setting, the application requires no modifications and can seamlessly utilize AsyncGBP$^+$. For hybrid and online computing settings, application adaptation follows a specific paradigm. Developers only need to utilize `Async_start_job` to encapsulate the original cryptographic operation APIs, and create a handler to appropriately process asynchronous notifications. For highly concurrent application scenarios, we recommend that application vendors choose callback-based asynchronous notifications, as they offer better performance compared to file descriptor-based methods.

VI. Evaluation

In this section, we conduct a series of experiments to validate the effectiveness of the cryptographic primitive optimization methods. We also evaluate AsyncGBP$^+$ using the OpenSSL built-in utility and compare its performance with other cryptographic acceleration schemes. Details of the experimental platform are presented in Table III.

TABLE III
EXPERIMENTAL PLATFORM CONFIGURATION

CPU	Intel Xeon Gold 5218R @2.10 GHz	RAM	32 GB
GPU	RTX 3070 (1.73 GHz, 5888 CUDA Cores)	OS	Ubuntu 20.04
Driver	NVIDIA Driver 510.108.03, CUDA 11.6	Software	OpenSSL 3.0.0

TABLE IV
KERNEL THROUGHPUT AND LATENCY OF KYBER768*

		Blocks	Threads/Block			
			32	64	128	256
Kyber. CPAPKE	KeyGen (kops/ms)	46	1189.95/1.24	1742.85/1.69	2131.95/2.78	1935.46/6.08
		92	1780.36/1.65	2187.64/2.69	2226.69/5.20	2018.05/11.78
		138	2033.60/2.17	2278.28/3.88	**2233.23**/8.19	2071.79/17.05
		184	2110.19/2.79	2220.40/5.30	2220.72/9.81	2115.47/22.27
	Enc (kops/ms)	46	646.07/2.28	899.81/3.27	1210.93/4.78	1046.96/10.04
		92	883.78/3.33	1214.08/4.85	1444.77/8.19	1226.14/18.96
		138	1030.94/4.28	1396.18/6.33	1364.94/12.94	1270.32/27.81
		184	1053.01/5.59	1447.72/8.13	**1474.03**/15.98	1311.26/35.92
	Dec (kops/ms)	46	1873.04/0.79	2451.96/1.20	**2984.78**/1.90	1684.80/6.65
		92	2147.88/1.37	2978.65/1.98	1903.27/6.19	1686.34/12.96
		138	2375.38/1.86	2041.54/4.33	1962.34/9.00	1866.07/18.93
		184	2110.55/2.79	1858.13/6.34	1989.31/11.84	1876.73/25.10
Kyber. CCAKEM	KeyGen (kops/ms)	46	1058.54/1.39	1519.46/1.94	1834.73/4.44	1697.77/6.93
		92	1582.47/1.86	1868.59/3.15	**1947.65**/5.95	1761.07/13.42
		138	1832.38/2.41	2017.53/4.38	1950.08/10.01	1812.22/19.49
		184	1832.05/3.21	1939.16/6.07	1775.26/11.49	1853.09/25.42
	Encaps (kops/ms)	46	309.47/4.76	513.93/5.73	930.18/6.01	1032.90/11.24
		92	516.18/5.70	800.02/7.36	**1415.47**/9.59	1186.84/20.66
		138	643.41/6.86	1127.83/7.83	1223.49/14.44	1150.12/30.72
		184	848.62/6.94	1222.13/9.64	1315.96/17.90	1251.75/37.63
	Decaps (kops/ms)	46	331.54/4.44	442.23/6.66	743.69/9.16	706.12/16.96
		92	506.77/5.81	644.42/9.14	788.39/17.94	731.31/33.27
		138	607.81/7.27	726.22/12.16	834.23/21.17	752.68/46.94
		184	563.01/10.46	792.42/14.86	**857.26**/27.47	765.22/61.56

* The maximum block size is limited to 256 by the scanning method [10], which requires the entire block to compute an NTT, with each warp processing $16 \times tiles\_per\_warp$ coefficients. $tiles\_per\_warp = ((n/2)/16)/(block\_size/32)$. In Kyber, n is 256, $tiles\_per\_warp$ can be 1, 2, 4, or 8. Hence, the number of threads per block can be 32, 64, 128, or 256.

TABLE V
KERNEL THROUGHPUT AND LATENCY OF X25519 AND ED25519

	Blocks	Threads/Block			
		256	512	768	1024
X25519 (kops/ms)	46	10757.51/1.10	**11728.68**/1.92	11195.13/3.16	10926.37/4.31
	92	10803.19/2.18	11181.33/4.21	11130.83/6.35	11111.11/8.48
	138	11259.79/3.14	11348.68/6.23	11144.61/9.51	11054.15/12.78
	184	11380.51/4.14	11346.82/8.30	11186.91/12.63	11094.30/16.98
Ed25519 SigGen (kops/ms)	46	21686.60/0.54	32472.98/0.73	27706.67/1.28	11080.67/4.25
	92	20141.76/1.17	34961.88/1.35	29482.85/2.40	13178.86/7.15
	138	20454.29/1.73	35451.09/1.99	29780.42/3.56	14896.47/9.49
	184	20698.58/2.28	**35832.52**/2.63	29557.84/4.78	15992.40/11.78
Ed25519 SigVer (kops/ms)	46	2428.27/4.85	4167.33/5.65	3946.25/8.95	3374.10/13.96
	92	3923.66/6.00	4417.87/10.66	4086.32/17.29	3453.68/27.28
	138	3825.58/9.24	**5209.27**/13.56	4525.93/23.42	3563.45/39.66
	184	4017.91/11.72	4926.46/19.12	4653.82/30.37	3658.12/51.51

TABLE VI
KERNEL THROUGHPUT AND LATENCY OF ECDSA-P256

	Blocks	Threads/Block			
		256	512	768	1024
ECDSA PureGPU SigGen (kops/ms)	46	15111.70/ 0.78	17774.34/1.33	18478.84/1.91	15711.98/5.21
	92	15016.73/1.57	17971.38/2.62	18470.80/3.83	16696.91/5.64
	138	14905.22/2.37	17997.02/3.93	18499.90/5.73	16468.52/8.58
	184	14888.54/3.16	17994.67/5.24	**18519.45**/7.63	16548.81/11.39
ECDSA SigGen r & k^{-1} (kops/ms)	46	15541.85/0.76	18258.04/1.29	18905.73/1.87	16786.21/2.81
	92	17138.20/1.37	18422.80/2.56	18953.77/3.73	17124.34/5.50
	138	17245.96/2.05	18499.59/3.82	18965.27/5.59	17352.76/8.14
	184	17541.77/2.69	18518.52/5.09	**18987.42**/7.44	17536.13/10.74
ECDSA PureGPU SigVer (kops/ms)	46	4654.69/2.53	5353.93/4.40	4391.71/8.04	3048.28/15.45
	92	4726.89/4.98	**5447.90**/8.65	5316.38/13.29	3145.47/29.95
	138	4580.42/7.71	5351.47/13.20	5337.86/19.86	3142.85/44.96
	184	4733.21/9.95	5372.19/17.54	5353.10/26.40	2981.37/63.20

A. Performance Evaluation of the Cryptographic Primitives

This subsection focuses on the performance of GPU kernel implementations of cryptographic primitives, which serve as the source of the overall framework performance.

Table IV presents the kernel performance of Kyber768 under different execution configurations. As the block size increases, latency shows a noticeable upward trend, while the throughput initially increases and then gradually declines. Although a larger grid size can boost throughput, this increase exhibits diminishing marginal returns. Moreover, the significant increase in latency is unacceptable for practical applications.

Table V presents the throughput of X25519 scalar multiplication and Ed25519 signature generation and verification, while Table VI illustrates the throughput of ECDSA-P256 signature generation and verification. The results represent that larger batch size benefits higher throughput, while the increase in latency remains acceptable. However, if the resource requirements exceed the threshold, variables spill into the GPU's local memory, causing performance degradation. Similar to the conclusions drawn from Table IV, a larger grid size can improve throughput, but it significantly increases latency, which is undesirable.

B. Performance Evaluation of AsyncGBP$^+$

This subsection presents the performance evaluation of AsyncGBP$^+$ under three working settings. In the online setting, we evaluate cryptographic operations such as X25519, signature generation and verification for ECDSA and Ed25519, as well as Kyber768 Keygen/Encaps/Decaps. The evaluation of the hybrid setting focuses on ECDSA signature generation. In the offline setting, we conduct experiments on Kyber768 key generation.

To ensure accurate and reliable results, we measure its performance using the command-line utility `openssl speed` [31]. This utility leverages the EVP APIs to execute a specified cryptographic algorithm repeatedly within a specified time range, then counts the completed operations and reports the throughput.

The previous analysis indicates that the callback-based asynchronous notification mechanism outperforms that based on file descriptors in high-concurrency scenarios. However, since the default `openssl speed` lacks support for this mechanism, we patched it to enable the callback-based asynchronous event notification. Additionally, we extended the `openssl speed` performance test for KEM to evaluate the performance of Kyber768 in AsyncGBP$^+$.

Table VII presents the performance of traditional cryptographic primitives in AsyncGBP$^+$ under a multi-process setting. To fully utilize the GPU's computing power, we enable

TABLE VII
PERFORMANCE OF TRADITIONAL CRYPTOGRAPHIC PRIMITIVES IN ASYNCGBP$^+$

		Online Setting					Hybrid Setting
		X25519 (kops)	Ed25519 SigGen (kops)	Ed25519 SigVer (kops)	ECDSA SigGen PureGPU† (kops)	ECDSA SigVer PureGPU† (kops)	ECDSA SigGen Hybrid‡ (kops)
Local Test*		9121.56	12398.76	4086.43	8729.76	4662.93	9880.43
AsyncGBP$^+$	1 Proc.	1143.20	1058.74	1052.88	773.92	572.62	1246.55
	2 Proc.	1873.46	1955.27	1843.12	1429.74	1048.26	2337.23
	4 Proc.	3283.40	3175.38	2782.03	2734.99	1761.71	4643.77
	6 Proc.	4509.69	5339.54	3432.10	4157.92	2331.74	6783.60
	8 Proc.	**5573.11**	**6333.08**	**3967.72**	**5053.65**	**2469.86**	**8684.55**

* "Local Test" refers to the testing process involving data transfer and kernel execution.
† "PureGPU" denotes that the operation of ECDSA is performed entirely on the GPU.
‡ "Hybrid" means the GPU computes r and k^{-1} required for ECDSA signature generation.

TABLE VIII
PERFORMANCE OF KYBER768 IN ASYNCGBP$^+$

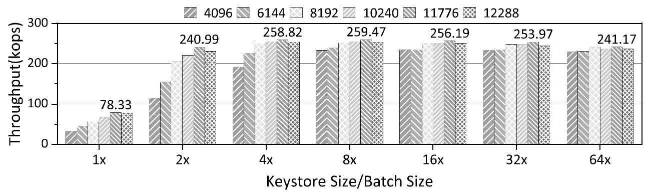
		Offline Setting	Online Setting		
		Kyber768 KeyGen (kops)	Kyber768 KeyGen (kops)	Kyber768 Encaps (kops)	Kyber768 Decaps (kops)
Local Test*		/	478.11	939.54	582.79
AsyncGBP$^+$	1 Proc.	258.51	217.31	480.20	374.46
	2 Proc.	495.38	393.96	847.44	672.58
	4 Proc.	780.36	639.85	1200.14	949.15
	6 Proc.	**1022.98**	**704.07**	1266.12	1074.98
	8 Proc.	992.86	595.76	**1295.00**	**1082.80**

* "Local Test" means the testing process involving data transfer and kernel execution.

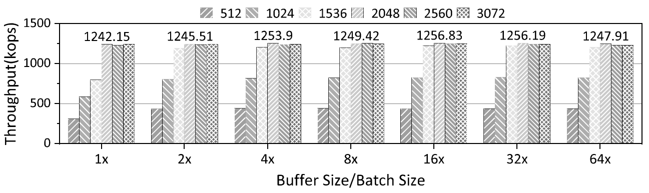
the NVIDIA Multiple Process Service (MPS) to run CUDA kernels concurrently on the same GPU. The results demonstrate that the performance of X25519, Ed25519 and ECDSA-P256 progressively improves as the number of processes increases. Compared with local tests of cryptographic primitives involving kernel execution and data copying, AsyncGBP$^+$ achieves 51.1%-97.1% of its performance.

Table VIII shows the performance of the Kyber768 in a multi-process setting with MPS enabled. Since Kyber's public and private keys as well as ciphertext sizes are much larger than those of traditional cryptography, data transfer significantly impacts the actual performance. By employing the stream technique, we observe a gradual performance improvement in Kyber, eventually surpassing local test performance. Furthermore, we notice that the offline setting consistently outperforms the online setting in the comparison of key generation, reaching between 1.19× and 1.67× the performance of the online setting.

In addition, we conduct a series of experiments to determine the optimal parameters for AsyncGBP$^+$ offline and hybrid settings. Firstly, we study the effect of keystore size and batch size on throughput, as illustrated in Fig. 8(a). We utilize the default test duration (10 seconds) of `openssl speed` for testing. According to the results, we determine the optimal batch size as 11776 and the keystore size as 11776 × 8 instances, which is approximately 215.6 MB, as analyzed in Section IV-B3. Based on the same testing procedure, we determine that the batch size for ECDSA signature generation in hybrid setting is 2048, with a buffer size of 2048 × 4 instances, which is 512 KB.



(a) Effect of keystore size and batch size on Kyber768 KeyGen throughput (offline setting).



(b) Effect of buffer size and batch size on ECDSA-P256 signature generation throughput (hybrid setting).

Fig. 8. Experiments on optimal parameters in AsyncGBP$^+$ offline and hybrid settings.

TABLE IX
PERFORMANCE COMPARISON WITH OTHER OPENSSL-COMPATIBLE ACCELERATORS ON TRADITIONAL CRYPTOGRAPHY

Cryptographic Accelerators	X25519 (kops)	Ed25519 SigGen (kops)	Ed25519 SigVer (kops)	ECDSA SigGen (kops)	ECDSA SigVer (kops)
OpenSSL Default Provider [17]*	23.90	24.32	7.64	38.04	12.76
Speedup† vs [17]	47.83×/ 233.18×	43.53×/ 260.41×	137.81×/ 519.34×	32.77×/ 228.30×	44.88×/ 193.56×
Intel QAT [32]	120	/	/	81.65	/
Speedup† vs [32]	9.52×/ 46.44×	/	/	15.27×/ 106.36×	/
Secure-IC SCZ_SP_BA452 [33]	1000	/	/	910	750
Speedup† vs [33]	1.14×/ 5.57×	/	/	1.37×/ 9.54×	0.76×/ 3.29×
Ours in single-process	1143.20	1058.74	1052.88	1246.55	572.62
Ours in multi-process	**5573.11**	**6333.08**	**3967.72**	**8684.55**	**2469.86**

* The experimental data is obtained in a single-process setting.
† Speedup represents the performance improvement of AsyncGBP$^+$ over other OpenSSL-compatible accelerators. Each cell shows two metrics: the performance improvement of AsyncGBP$^+$ in both single-process and multi-process environments, separated by a slash (/).

C. Comparison With Other OpenSSL-Compatible Accelerators

This subsection compares AsyncGBP$^+$ with other OpenSSL-compatible cryptographic accelerators, including CPU-based, ASIC-based, and FPGA-based schemes.

Table IX presents the performance comparison between AsyncGBP$^+$ and other OpenSSL-compatible cryptographic accelerators on traditional cryptography. Our work demonstrates significantly higher performance than the OpenSSL default provider. The performance of AsyncGBP$^+$ is improved by 32.77× to 137.81× under the single-process setting, with even greater improvements under the multi-process setting. Intel QuickAssist Technology (QAT) is an ASIC-based cryptographic acceleration solution. Ref [32] evaluated the performance of QAT using `openssl speed`, where the

TABLE X
PERFORMANCE COMPARISON WITH OTHER OPENSSL-COMPATIBLE ACCELERATORS ON POST-QUANTUM CRYPTOGRAPHY

	Kyber768 KeyGen (kops)	Kyber768 Encaps (kops)	Kyber768 Decaps (kops)
oqs-provider C [34]*	13.46	11.43	9.55
Speedup† vs [34]	16.14×/76.00×	42.01×/113.30×	39.21×/113.38×
oqs-provider AVX [34]*	47.52	46.76	59.63
Speedup vs [34]	4.57×/21.53 ×	10.27×/27.69×	6.28×/18.16×
Ours in single-process	217.31	480.20	374.46
Ours in multi-process	**1022.98**	**1295.00**	**1082.80**

* The performance of oqsprovder 0.5.2 in single-process setting.
† Speedup represents the performance improvement of AsyncGBP$^+$ over other OpenSSL-compatible accelerators. Each cell shows two metrics: the performance improvement of AsyncGBP$^+$ in both single-process and multi-process environments, separated by a slash (/).

throughput of X25519 is 120 kops. In comparison, our implementation of X25519 presents significantly higher performance, outperforming Intel QAT by an order of magnitude. Ref [33] described an FPGA-based accelerator for TLS handshake. Its ECDSA-P256 signature performance is 910 kops, while the throughput of AsyncGBP$^+$ outperforms this work by 7.0× in a multi-process setting.

Table X shows the performance comparison of AsyncGBP$^+$ with other OpenSSL-compatible cryptographic accelerators on post-quantum cryptography. Oqs-provider [34] is an OpenSSL provider developed by the Open Quantum Safe project based on `liboqs`. We evaluate the performance of Kyber768 of oqs-provider on the same testing platform. In multi-process tests, AsyncGBP$^+$ shows a substantial performance improvement, ranging from 76.00× to 113.30× compared to the C implementation, and from 18.16× to 27.69× compared to the AVX implementation.

D. Discussion

In this section, we discuss countermeasures to side-channel attacks and the scalability of the proposed framework.

1) Side-Channel Attack: The cryptographic primitives used in AsyncGBP$^+$ are implemented to mitigate the side-channel attacks. For X25519 and EdDSA, the curve-level security is primarily ensured through the curve itself [3]. For ECDSA, any fixed-point multiplication is conducted with fixed number times of point additions via the pre-computation technique, preventing the leakage of random number k (and private key d) through side-channel attacks. For Kyber, secret-related branches and memory accesses are avoided in implementation. We optimize NTT operations using Tensor Cores, which can be viewed as atomic instruction units and are almost impossible to be impacted by timing attacks [35]. Moreover, the multi-precision representation adopted in our paper serves as a form of arithmetic masking, which can be considered as a measure to prevent side-channel attacks. More details can be found in [9], [10], [29].

Furthermore, the framework design of AsyncGBP$^+$ is inherently resistant to side-channel attacks. Firstly, cryptographic computations are performed entirely on the backend. This introduces significant noise into the measurable latency for Internet attackers, thereby substantially increasing the difficulty of side-channel attacks, even in the online setting of AsyncGBP$^+$. Secondly, hybrid and offline settings are introduced to decouple cryptographic computations from specific requests. This design not only boosts the performance of cryptographic services but also makes it more challenging for attackers to conduct side-channel attacks.

2) Scalability:

Support for multiple CPUs/GPUs. AsyncGBP$^+$ is fully compatible with the OpenSSL Provider mechanism. Given OpenSSL's support for multi-threading [36], our design can also run effectively in multi-CPU environments. Moreover, the underlying cryptographic primitives of AsyncGBP$^+$ are built based on CUDA, which offers a series of device management APIs to facilitate program execution on multiple GPUs [37], such as `cudaGetDeviceCount` and `cudaSetDevice`. These capabilities enable our framework to be adapted to multi-GPU settings with minimal modifications.

Support for AMD and Intel GPUs. From an architectural perspective, AsyncGBP$^+$ features universality, naturally supporting other types of high-throughput computing units, including AMD and Intel GPUs. Note that AsyncGBP$^+$ is not inherently bound to CUDA. A potential approach is to directly use OpenCL to implement cryptographic primitives for AMD and Intel GPUs, and then integrate these with AsyncGBP$^+$. Furthermore, as both AMD and Intel provide tools to port CUDA programs to their ROCm [38] and OpenAPI [39] platforms, an alternative approach is using these tools to migrate AsyncGBP$^+$ (more specifically, its cryptographic primitives) to AMD and Intel GPUs. This requires minor adaptation work for the implementation of cryptographic primitives.

VII. CONCLUSION

In this paper, we propose AsyncGBP$^+$, a heterogeneous and high-performance TLS acceleration framework that bridges the gap between the OpenSSL provider mechanism and GPUs. AsyncGBP$^+$ highlights the potential of GPU-based cryptographic accelerators to enhance the throughput of SSL/TLS in high-traffic environments, regardless of whether the workload is based on traditional or post-quantum cryptography. In addition, a practical implementation guide is provided for application vendors to update their systems. Our future work includes enhancing support for post-quantum cryptography and evaluating the performance in large-scale TLS concurrency scenarios.

REFERENCES

[1] "Transport layer security," Accessed: May 13, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Transport_Layer_Security
[2] "SSL pulse." Accessed: May 13, 2024. [Online]. Available: https://www.ssllabs.com/ssl-pulse
[3] A. Langley, M. Hamburg, and S. Turner, "RFC 7748: Elliptic curves for security," Internet Eng. Task Force, Tech. Rep., 2016. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7748
[4] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ECDSA)," *Int. J. Inf. Secur.*, vol. 1, pp. 36–63, Aug. 2001.

[5] S. Josefsson and I. Liusvaara, "Edwards-curve digital signature algorithm (EdDSA)," Internet Res. Task Force, Tech. Rep., 2017. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8032

[6] W. N. Chelton and M. Benaissa, "Fast elliptic curve cryptography on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 2, pp. 198–205, Feb. 2008.

[7] O. Hajihassani, S. K. Monfared, S. H. Khasteh, and S. Gorgin, "Fast AES implementation: A high-throughput bitsliced approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 10, pp. 2211–2222, Oct. 2019.

[8] J. Dong, F. Zheng, J. Lin, Z. Liu, F. Xiao, and G. Fan, "EC-ECC: Accelerating elliptic curve cryptography for edge computing on embedded GPU TX2," *ACM Trans. Embedded Comput. Syst.*, vol. 21, no. 2, pp. 1–25, 2022.

[9] L. Gao, F. Zheng, N. Emmart, J. Dong, J. Lin, and C. Weems, "DPF-ECC: Accelerating elliptic curve cryptography with floating-point computing power of GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2020, pp. 494–504.

[10] L. Wan et al., "A novel high-performance implementation of CRYSTALS-kyber with AI accelerator," in *Proc. Eur. Symp. Res. Comput. Secur.,* Copenhagen, Denmark: Springer, 2022, pp. 514–534.

[11] N. Gupta, A. Jati, A. K. Chauhan, and A. Chattopadhyay, "PQC acceleration using GPUs: Frodokem, newhope, and kyber," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 575–586, Mar. 2021.

[12] A. Ahmadzadeh, O. Hajihassani, and S. Gorgin, "A high-performance and energy-efficient exhaustive key search approach via GPU on DES-like cryptosystems," *J. Supercomput.*, vol. 74, no. 1, pp. 160–182, Jan. 2018.

[13] C. Tezcan, "Key lengths revisited: GPU-based brute force cryptanalysis of DES, 3DES, and PRESENT," *J. Syst. Archit.*, vol. 124, Mar. 2022, Art. no. 102402.

[14] Y. Gao, J. Xu, and H. Wang, "CuNH: Efficient GPU implementations of post-quantum KEM NewHope," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 3, pp. 551–568, Mar. 2022.

[15] J. Bos et al., "CRYSTALS-Kyber: A CCA-secure module-lattice-based KEM," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS & P)*, 2018, pp. 353–367.

[16] "Protecting chrome traffic with hybrid kyber KEM," Accessed: May 13, 2024. [Online]. Available: https://blog.chromium.org/2023/08/protecting-chrome-traffic-with-hybrid.html

[17] "OpenSSL." Accessed: May 13, 2024. [Online]. Available: https://www.openssl.org/

[18] "OpenSSL 3.0.0 design," Accessed: May 13, 2024. [Online]. Available: https://www.openssl.org/docs/OpenSSL300Design.html

[19] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," Internet Eng. Task Force, Tech. Rep., 2018. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8446

[20] F. PUB, "Digital signature standard (DSS)," *Fips pub.*, pp. 186–192, 2000.

[21] A. Langlois and D. Stehlé, "Worst-case to average-case reductions for module lattices," *Designs, Codes Cryptogr.*, vol. 75, no. 3, pp. 565–599, 2015.

[22] E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes," in *Proc. Annu. Int. Cryptol. Conf.,* Berlin, Heidelberg: Springer, 1999, pp. 537–554.

[23] R. Avanzi et al., "CRYSTALS-kyber algorithm specifications and supporting documentation," *NIST PQC Round*, vol. 2, pp. 1–43, 2021.

[24] "ASYNC_start_job," Accessed: May 13, 2024. [Online]. Available: https://www.openssl.org/docs/manmaster/man3/ASYNC_start_job.html

[25] "provider-keymgmt," Accessed: May 13, 2024. [Online]. Available: https://www.openssl.org/docs/manmaster/man7/provider-keymgmt.html

[26] "provider-kem," Accessed: May 13, 2024. [Online]. Available: https://www.openssl.org/docs/manmaster/man7/provider-kem.html

[27] H. Krawczyk and P. Eronen, "HMAC-based extract-and-expand key derivation function (HKDF)," Internet Engineering Task Force, Tech. Rep., 2010. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5869

[28] "NVIDIA tensor cores unprecedented acceleration for HPC and AI," Accessed: May 13, 2024. [Online]. Available: https://www.nvidia.com/en-us/data-center/tensor-cores/

[29] W. Pan, F. Zheng, Y. Zhao, W.-T. Zhu, and J. Jing, "An efficient elliptic curve cryptography signature server with GPU acceleration," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 1, pp. 111–122, Jan. 2017.

[30] D. Hankerson, A. Menezes, and S. V. Springer, *Guide to Elliptic Curve Cryptography.* Berlin, Heidelberg: Springer, 2004.

[31] "openSSL-speed," Accessed: May 13, 2024. [Online]. Available: https://www.openssl.org/docs/manmaster/man1/openssl-speed.html

[32] "Building software acceleration features in the intel quick assist technology engine for OpenSSL 1.1.1," Accessed: May 13, 2024. [Online]. Available: https://web.archive.org/web/20230609034300/ https://www.intel.com/content/www/us/en/developer/articles/guide/building-software-acceleration-features-in-the-intel-qat-enginefor-openssl.html

[33] "TLS handshake hardware accelerator," Secure-IC. Accessed: May 13, 2024. [Online]. Available: https://www.secure-ic.com/wp-content/uploads/2022/11/SCZ_SP_BA452_TLS_Handshake_Hardware_Accelerator_Product_Sheet_Web.pdf

[34] "OQS-OpenSSL: OpenSSL 3 provider containing post-quantum algorithms," Accessed: May 13, 2024. [Online]. Available: https://github.com/open-quantum-safe/oqs-provider

[35] T. Nakai, D. Suzuki, and T. Fujino, "Timing black-box attacks: Crafting adversarial examples through timing leaks against DNNs on embedded devices," *IACR Trans. Cryptogr. Hardw. Embedded Syst.*, pp. 149–175, 2021.

[36] "CRYPTO_THREAD_run_once," OpenSSL. Accessed: May 13, 2024. [Online]. Available: https://www.openssl.org/docs/manmaster/man3/CRYPTO_THREAD_run_once.html

[37] "Device management, " NVIDIA. Accessed: May 13, 2024. [Online]. Available: https://docs.nvidia.com/cuda/archive/11.4.0/cuda-runtime-api/group__CUDART__DEVICE.html

[38] "HIP porting guide, " AMD. Accessed: May 13, 2024. [Online]. Available: https://rocm.docs.amd.com/projects/HIP/en/docs-5.7.1/user_guide/hip_porting_guide.html

[39] "Migrate from CUDA* to C++ with SYCL*," Intel. Accessed: May 13, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/training/migrate-from-cuda-to-cpp-with-sycl.html

[40] Y. Bian et al. "AsyncGBP: Unleashing the potential of heterogeneous computing for SSL/TLS with GPU-based provider," in *Proc. 52nd Int. Conf. Parallel Process.*, 2023, pp. 337–346.

Yi Bian received the B.E. degree from the School of Information Science and Engineering, Shandong University, Jinan, China, in 2018. He is currently working toward the Ph.D. degree with the School of Computer Science and Technology, University of Chinese Academy of Science, Beijing, China. His research interests include applied cryptography and high-performance computing.

Fangyu Zheng received the B.E. degree in information security from the University of Science and Technology of China, Hefei, China, in 2011, and the Ph.D. degree in information security from the University of Chinese Academy of Sciences, Beijing, China, in 2016. Currently, he is an Associate Professor with the School of Cryptology, University of Chinese Academy of Sciences. His research interests include applied cryptography and high-performance computing.

Yuewu Wang received the Ph.D. degree from the Graduate University, Chinese Academy of Sciences, Beijing, China, in 2008. Currently, he is a Professor with the School of Cryptology, Chinese Academy of Sciences. His research interests focus on information security, including mobile security, network attack simulation, and information system security evaluation.

Lingguang Lei received the Ph.D. degree in computer science from the University of Chinese Academy of Sciences, Beijing, China, in 2013. Currently, she is an Associate Professor with the Institute of Information Engineering, Chinese Academy of Sciences. Her research interests focus on information security, including mobile security, container security, network security, and information system security evaluation.

Yuan Ma received the B.E. degree from Tsinghua University, in 2009, and the Ph.D. degree from the University of Chinese Academy of Sciences, in 2014. Currently, he is an Associate Professor with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing. His research interests include physical random number generator design and evaluation, cryptographic algorithm high-speed implementation, and hardware security modules.

Tian Zhou received the B.E. degree from Hangzhou Dianzi University. He is currently working toward the Ph.D. degree with the University of Science and Technology of China. His research interest includes applied cryptography.

Jiankuo Dong received the B.E. degree in computer science and technology from Xi'an Jiaotong University, Xi'an, China, in 2014, and the Ph.D. degree in cyberspace security from the University of Chinese Academy of Sciences, Beijing, China, in 2019. Currently, he is an Associate Professor with the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, China. His research interests include public key cryptography and applied cryptography.

Guang Fan received the B.E. degree from Beijing Institute of Technology, Beijing, China, in 2018, and the Ph.D. degree in cyberspace security from the University of Chinese Academy of Science, Beijing, China, in 2023. Currently, he is an Assistant Researcher with Ant Group. His research interests include privacy preserving computing and high performance computing.

Jiwu Jing (Member, IEEE) received the B.E. degree from the Department of Electronics Engineering, Tsinghua University, Beijing, China, in 1987, and the M.Sc. and Ph.D. degrees from the Graduate School, Chinese Academy of Sciences, Beijing, in 1990 and 2003, respectively. Currently, he is a Professor with the School of Cryptology, University of Chinese Academy of Sciences. His main research interest is information security, including public key infrastructure, identity management, and fault tolerance.