AsyncGBP: Unleashing the Potential of Heterogeneous Computing for SSL/TLS with GPU-based Provider

Yi Bian
School of Computer Science and Technology, University of Chinese Academy of Sciences
bianyi18@mails.ucas.ac.cn

Fangyu Zheng*
School of Cryptography, University of Chinese Academy of Sciences
zhengfy1028@hotmail.com

Yuewu Wang
School of Cryptography, University of Chinese Academy of Sciences
wangyuewu@ucas.ac.cn

Lingguang Lei
State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
leilingguang@iie.ac.cn

Yuan Ma
State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
mayuan@iie.ac.cn

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

Jiwu Jing
School of Cryptography, University of Chinese Academy of Sciences
jwjing@ucas.ac.cn

ABSTRACT

The proliferation of IoT and 5G technologies has led to an explosion of data traffic that data centers must handle while ensuring secure transmission via SSL/TLS. The high volume of cryptographic operations required imposes performance bottlenecks. The GPU-based cryptographic accelerator is one of the competitive solutions. However, significant structural differences with practical applications confine their capacities to specific domains, such as offline cryptanalysis, undermining their potential for real-world cryptographic acceleration.

This paper investigates the feasibility of using GPUs as cryptographic accelerators for concurrent data secure transmission scenarios like SSL/TLS. Specifically, we propose AsyncGBP, a framework that integrates the original OpenSSL software stack with the heterogeneous GPU-based accelerator. To enhance user-friendliness and take full advantage of GPUs' SIMT execution model, AsyncGBP features an OpenSSL-compatible asynchronous design, which seamlessly converts cryptographic requests from synchronous to asynchronous mode, efficiently aggregates numerous requests, and rationally schedules GPU for computation. We also provide a fine-grained GPU-based cryptographic algorithm stack that includes X25519, Ed25519, and ChaCha20-Poly1305. A comprehensive evaluation shows that AsyncGBP can efficiently achieve up to 97% of GPU local performance on an RTX 3070, resulting in an improvement of up to 137x compared to the default OpenSSL provider in a single-process setting. Furthermore, AsyncGBP outperforms the existing fastest commercial-off-the-shelf OpenSSL-compatible TLS accelerator by a significant margin, achieving a 5.3x to 7.0x performance improvement.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Security and privacy** → *Digital signatures*; *Block and stream ciphers.*

KEYWORDS

TLS 1.3, Heterogeneous Computing, Graphics Processing Unit

ACM Reference Format:
Yi Bian, Fangyu Zheng, Yuewu Wang, Lingguang Lei, Yuan Ma, Jiankuo Dong, and Jiwu Jing. 2023. AsyncGBP: Unleashing the Potential of Heterogeneous Computing for SSL/TLS with GPU-based Provider. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605620

1 INTRODUCTION

Secure Socket Layer (SSL) and Transport Layer Security (TLS) are cryptographic protocols designed to provide secure communication over the Internet, ensuring the confidentiality, integrity, and authenticity of the data. SSL/TLS has become the de facto standard for securing Internet communication and is widely used in e-commerce, online banking, and other sensitive online transactions. According to the SSL Pulse project by the security company Qualys [19], as of April 2023, 99.9% of the popular websites worldwide support SSL/TLS, and 38.2% and 61.7% of them choose TLS 1.2 and TLS 1.3 as the best practice for SSL/TLS protocols [19], respectively.

*Fangyu Zheng is the corresponding author.

ICPP 2023, August 07–10, 2023, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0843-5/23/08.
https://doi.org/10.1145/3605573.3605620

SSL/TLS is built upon computation-intensive cryptographic algorithms. Public key cryptography is used 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. X25519 [13] and Ed25519 [11] are currently recommended in TLS 1.3 for key exchange and digital signature due to their relatively high computational speed and resistance against various attacks such as differential power analysis, side-channel attacks, and fault attacks. For the symmetric cipher suites of TLS 1.3, ChaCha20-Poly1305 and AES (with CCM or GCM) are the only two Authenticated Encryption with Associated Data (AEAD) algorithms. Although AES is currently the primary option for many servers, ChaCha20-Poly1305 has become the choice for new lightweight computing platforms due to its fast encryption speed and security advantages [21, 22]. With the increasing popularity of the Internet of Things (IoT), the application field of ChaCha20-Poly1305 is expected to further expand, leading to a significant increase in data.

1.1 Motivations

The underlying cryptographic primitives used in SSL/TLS protocol have a high computational complexity, which presents significant performance challenges when protecting large amounts of traffic data. In order to minimize the performance overhead imposed by cryptographic operations, it is common practice in the industry to employ hardware accelerators to perform computationally intensive cryptographic operations. Hardware cryptography accelerators are generally based on hardware platforms such as ASICs, GPUs, and FPGAs [5, 8, 9, 26, 28], which offload time-consuming cryptographic operations from CPUs to dedicated hardware, thereby reducing the resource consumption of data centers. GPU-based cryptographic accelerators can act as coprocessors of CPUs to handle cryptographic operations exclusively. Studies [9, 26] have demonstrated that this solution can achieve high throughput with significant advantages.

However, previous studies in this area have mostly been limited to experiments or proof-of-concept level [9, 25, 27], without practical deployment in real-world production environments. Many of these works focused on pure cryptographic algorithm implementations, without considering their real-world workhorse delivery [9, 27], while others focused on cryptanalysis (e.g., brute-force attack) using GPUs [1, 25]. These efforts still face technical challenges before they can be deployed as real-world cryptographic accelerators, such as the incompatibility between the parallel computing mode of GPUs and the serial mode of OpenSSL which requires architectural adaptations. 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.

1.2 Contributions

To overcome the limitations of prior studies, we propose AsyncGBP (**Async**hronous **G**PU-**B**ased **P**rovider), a framework that harnesses GPUs to accelerate cryptographic algorithms while ensuring complete compatibility with OpenSSL. This enables easy integration of the powerful cryptographic workhorse into existing systems. Specifically, we make the following contributions:

- Firstly, we analyze the working mechanism of the OpenSSL 3.0 provider and explore the feasibility of bringing the powerful GPU-based workhorse to real-world OpenSSL application scenarios. An OpenSSL-compatible GPU-based cryptographic primitive acceleration framework, AsyncGBP, is proposed, which realizes the conversion between synchronous and asynchronous cryptographic operations based on the available resources (e.g., `ASYNC_JOB`), efficient data aggregation, and reasonable GPU scheduling.
- Secondly, we optimize the performance of SSL/TLS cryptographic primitives on GPUs. Specifically, we have focused on optimizing the usage of instructions, memory access patterns, asynchronous data transfer mechanisms, and parallel execution modes. Our optimizations for ChaCha20-Poly1305 have achieved an impressive throughput of 672.64 Gbps on the NVIDIA RTX 3070, which is the currently known maximum throughput for this algorithm.
- Finally, we implement a prototype of AsyncGBP on NVIDIA RTX 3070. A series of comprehensive experiments with the command-line utility `openssl speed` demonstrates that AsyncGBP can bridge synchronous mode and asynchronous mode to take full advantage of its SIMT feature. The empirical results show that AsyncGBP can efficiently output up to 97% of GPU's local performance in RTX 3070, resulting in an improvement of 2.59-137.81 times compared to the default OpenSSL provider.

To the best of our knowledge, this is the first work that successfully bridges the gap between GPU-based cryptographic acceleration and the OpenSSL provider mechanism, resulting in significant speed improvements. AsyncGBP outperforms the existing fastest OpenSSL-compatible TLS accelerator by a significant margin, demonstrating the effectiveness of the proposed design and the potential of GPU-based cryptographic accelerators to enhance the throughput of SSL/TLS in high-traffic scenarios such as cloud computing centers.

The rest of the paper is organized as follows. Section 2 presents the background and required knowledge. Section 3 illustrates the overall framework of AsyncGBP. Section 4 details the optimization scheme for GPU-based cryptographic primitives and implementation of AsyncGBP. In Section 5, we conduct a series of experiments and compare them with related works. Finally, we conclude the paper in Section 6.

2 PRELIMINARY

This section introduces the TLS 1.3 protocol, OpenSSL provider mechanism, and asynchronous mode. In addition, the basic theory of ChaCha20-Poly1305, X25519, and Ed25519 is illustrated.

2.1 TLS 1.3, OpenSSL Provider and ASYNC_JOB

Transport Layer Security (TLS) is a protocol to protect online communications. TLS 1.3 [20] proposed in 2018 is the latest version, which enhances the security and performance of the protocol, streamlines the handshake protocol, removes weak cryptographic

algorithms and adds advanced algorithms to provide better security, such as ChaCha20-Poly1305, X25519, and Ed25519.

As an open-source cryptographic library, OpenSSL is widely used for secure communication using the SSL/TLS [17]. In order to transparently utilize third-party cryptographic implementations, OpenSSL has proposed the engine and provider mechanisms and considers the provider as a focus for the future [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 module or as a dynamically loaded module. The `Core` is a component in `libcrypto` that enables applications to utilize algorithm implementations from providers.

OpenSSL supports asynchronous mode, implemented by the fiber-based `ASYNC_JOB` [16], 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 request has been delivered to the accelerator. Two methods are available to inform the applications about the job resumption: file descriptor or callback. When receiving a 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.

2.2 ChaCha20-Poly1305

ChaCha20 is an ARX-based stream cipher, one of the cipher suites adopted by TLS 1.3 [20], which is widely used in mobile devices, web browsers, and popular websites. It takes a 256-bit key and a 96-bit nonce as input and performs 20 rounds in counter mode to generate a key stream XORed with the plaintext. The core component of ChaCha20 is the quarter-round function, with four of these functions comprising a round. Poly1305, on the other hand, is a message-authentication code (MAC) designed by D. J. Bernstein [2]. It takes a 32-byte one-time key and a variable-length message as input to generate a 16-byte tag using finite field arithmetic over prime $(2^{130} - 5)$.

ChaCha20-Poly1305 is an authenticated encryption with additional data (AEAD) algorithm that combines the ChaCha20 stream cipher with the Poly1305 message authentication code to provide confidentiality and integrity protection for data. In the ChaCha20-Poly1305 structure, the 32-byte one-time key of Poly1305 is generated using the ChaCha20 stream cipher.

2.3 Curve25519 and Edwards25519

Curve25519 is a Montgomery curve [14] proposed by Daniel J. Bernstein in 2006 [3], defined on the 255-bit prime field $GF(p_{25519})$, where $p_{25519} = 2^{255} - 19$. Curve25519 is defined by the equation (1). RFC 7748 [13] released by IRTF in 2016 regards Curve25519 as the recommended curve and proposes that X25519 can be applied in Elliptic Curve Diffie-Hellman (ECDH) protocol. X25519 takes scalar k and the x-coordinate of point P as inputs and produces the x-coordinate of $[k]P$ as output. Therefore, the main workload of this function is unknown point scalar multiplication over Curve25519.

$$y^2 = x^3 + 486662x^2 + x \tag{1}$$

Edwards introduced a normal form for elliptic curves in [7], which are defined by $x^2+y^2 = c^2+c^2x^2y^2$. Bernstein et al. introduced twisted Edwards curves[4], and Edwards25519 is defined by the equation (2).

$$-x^2 + y^2 = 1 + \frac{121665}{121666}x^2y^2 \tag{2}$$

RFC 8032 [11] proposes Ed25519, which is a signature scheme that uses Edwards25519 as a recommended parameter for EdDSA and provides 128-bit security strength. There are two main variants of EdDSA, which differ in whether the pre-hashing is an identity function (PureEdDSA) or a collision-resistant hash function (HashEdDSA). The EdDSA used in TLS 1.3 is the former, so our focus is on PureEdDSA. Algorithm 1 show the process of Ed25519 and signature generation.

Algorithm 1: Ed25519 Signature Generation

Input: message M of arbitrary size, 32-octet private key d, the public key A
Output: 64-octet signature (R, S)
1 h=SHA-512(d);
2 s=$h_{[0...255]}$;
3 r=SHA-512($h_{[256...511]}||M$)mod L (L is the order of Edwards25519);
4 R=Encode ($[r]G$);
5 k=SHA-512($R||A||M$);
6 S=$(r + ks)$ mod L;
7 **return** (R, S);

For the signature verification, the signature is valid if the equation (3) holds, where k=SHA-512($R||A||M$).

$$[2^3][S]G = [2^3]Decode(R) + [2^3][k]Decode(A) \tag{3}$$

3 OVERVIEW

To achieve efficient TLS acceleration for applications, we propose AsyncGBP, a high-performance asynchronous TLS acceleration framework that leverages GPUs as accelerators to offload cryptographic operations. Its main ideas are also applicable to other types of SIMT cryptographic accelerators. In this section, we illustrate the design goals, challenges, and overall architecture of AsyncGBP.

3.1 Design Goals

As we desire to propose a cost-effective solution for integrating heterogeneous accelerators into existing applications, rather than rebuilding systems from scratch, the following requirements need to be taken into consideration.

- **Easy-integration**. Application vendors can easily integrate high-performance TLS acceleration services into their systems with minimal migration costs.
- **High-performance**. The GPU-based heterogeneous accelerator should deliver high-throughput and acceptable latency implementations of cryptographic primitives for the TLS protocol.
- **Scalability**. The framework needs to adapt to different scales of cryptographic computation requests and easily integrate additional cryptographic primitives in the future.

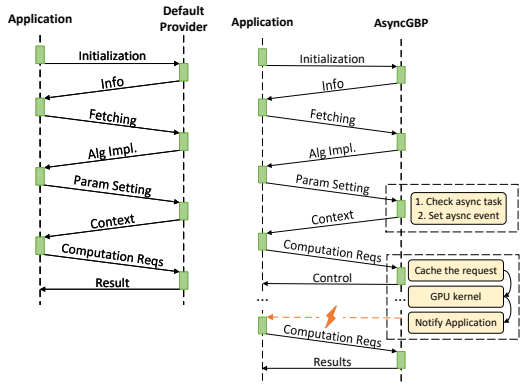
3.2 Technical Challenges

Although GPUs have inherent advantages in computing power, our primary goal is to leverage them in practical cryptographic scenarios and reduce porting costs. Therefore, accomplishing optimal performance of the overall system in this context presents numerous practical challenges.

Architectural Adjustments. There are significant differences in the working modes of GPUs and OpenSSL, especially in terms of 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 facilitate the seamless integration of working modes, the architecture needs to be designed to bridge the gap between GPUs and OpenSSL.

The Balance between User Experience and Performance. To ensure that the implementation of the underlying cryptographic primitives is transparent to upper-level applications, a balance needs to 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, it is necessary to adopt the standard interface to ensure ease of use while minimizing performance degradation.

Practical GPU-based Implementation of Cryptographic Primitives. The previous works focus on theoretical research, mainly concerning the running time of the kernel, which may be applied to algorithmic brute-force attacks. However, more attention is paid to the effective output of GPU computing power in practical applications, especially in the TLS request processing scenario, which requires the close integration of data transmission, kernel scheduling, and implementation of cryptographic primitives. Therefore, it is important to reasonably deploy both software and hardware resources to achieve high-performance implementation of cryptographic primitives.



(a) Default Workflow. **(b) AsyncGBP Workflow.**

Figure 1: Provider Workflow.

3.3 Workflow Analysis

Although the provider offers many different implementations of cryptographic primitives, the primary workflow can be summarized into four general stages, as shown in Figure 1a.

(1) **Stage 1: Provider Loading and Initialization.** In this stage, an application explicitly or implicitly invokes the provider initialization API. The component `Core` loads the provider into the memory, calls the entry point for initialization, and obtains the basic information of the provider, including algorithm implementations.
(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. Initially, the cache is searched. If the first search fails, `Core` initiates a query to the provider and caches the result.
(3) **Stage 3: Algorithm Parameters Setting.** Parameters related to cryptographic computations, such as algorithm types, operation types, and keys, are set during this stage.
(4) **Stage 4: Cryptographic Computation.** At this stage, the application initiates a cryptographic computation request using high-level interfaces (i.e., EVP APIs) and obtains results. This stage 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 reset or not 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.

Figure 1b illustrates the workflow of our proposed AsyncGBP. To ensure compatibility with OpenSSL, the first two stages are consistent with the default workflow, while the last two stages are modified to bridge the differences between OpenSSL and GPUs.

The Modifications in Stage 3. Synchronous cryptographic requests block the application while waiting for results, which is incompatible with the working mode of GPUs. Therefore, asynchronous cryptographic requests are essential. To support this functionality, we implement a series of measures in stage 3, including establishing asynchronous events when receiving a request, switching the context after caching the request, and notifying the application once the cryptographic operation is complete.

The Modifications in Stage 4. Batch processing is essential for optimizing GPU performance. To achieve this objective, it is necessary for stage 4 to cache a certain number of cryptographic requests before performing any cryptographic operations. Furthermore, GPUs need to be efficiently scheduled to perform various cryptographic operations. Once the computation is complete, the application should be notified to obtain results through the previously registered asynchronous event.

3.4 Overall Architecture

Figure 2 illustrates the workflow of AsyncGBP in the Cryptographic Computation stage (Stage 4), which comprises three steps: Request Sending, Cryptographic Computation, and Result Retrieval. The overall architecture of AsyncGBP is divided into three layers, with the functionality of each layer introduced in the corresponding step. The colored rectangles represent the modules that are currently active, while the gray rectangles indicate the inactive modules

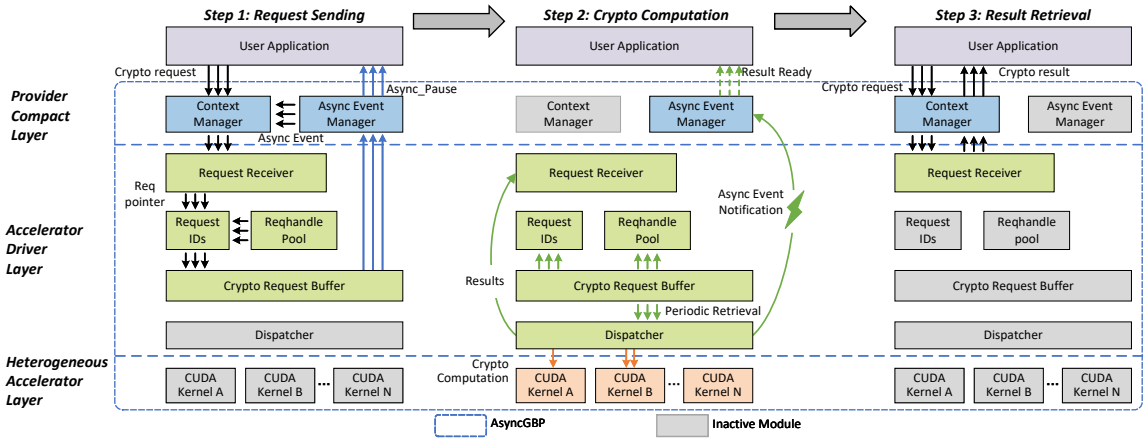


Figure 2: An overview of AsyncGBP in the Cryptographic Computation Stage.

during this step. It should be noted that the provider workflow's first three stages are not presented in the figure as they are not computationally intensive.

The asynchronous nature of the framework supports the concurrent execution of cryptographic requests. Consequently, the actual execution process may not strictly adhere to that depicted in the figure. For instance, while some cryptographic requests are being cached, others may undergo either crypto computation or result retrieval processes concurrently.

Step 1: Request Sending. 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 Provider Compatibility Layer (*ProvCompat*). The incoming parameters are bound to a context by the Context Manager (CM). After binding, the Asynchronous Event Manager (AEM) creates an asynchronous event to notify the application when the computation completes. Then, the context is delivered to the Accelerator Driver Layer (*AccDrv*) by the Request Receiver.

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 request type. By leveraging the lightweight request caching technology, we successfully achieve the goal of converting requests between synchronous and asynchronous modes and aggregating cryptographic requests.

Step 2: Crypto Computation. *AccDrv* comprises a crucial component for cryptographic computation, namely the dispatcher, which is responsible for two primary functions: scheduling GPU processing functions to perform cryptographic computation and notifying callers through asynchronous events. Specifically, the dispatcher periodically checks the ring buffer to identify any requests requiring processing. If there are no pending requests, the dispatcher enters a sleep state and awaits the subsequent polling cycle. Conversely, if any pending requests exist, the dispatcher fetches them and invokes the GPU processing function associated with the algorithm type.

As the ring buffer only stores request IDs, the dispatcher needs to obtain the cryptographic requests through the ⟨*ID, ReqHandle, Request*⟩ chain. The actual cryptography operations take place in Heterogeneous Accelerator Layer. Specifically, the dispatcher calls corresponding GPU kernels based on the request type to perform cryptographic computations, with the results binding to the corresponding *ReqHandle* through the chain.

Since the computation process is asynchronous, the application remains unware of the completion time for cryptographic computations. Therefore, it is essential for the dispatcher to notify the application to retrieve the calculation result through an AEM-registered asynchronous event. After the computation is complete, the request IDs and *ReqHandles* are reinitialized in preparation for subsequent use.

Step 3: Result Retrieval. When notified of an asynchronous event, 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. *ProvCompat* can return the results directly, thereby ending the entire cryptographic computation process.

4 HETEROGENEOUS PROVIDER IMPLEMENTATION

In this section, we present the implementation details of the proposed scheme in a bottom-up manner. Firstly, we introduce the optimizations of cryptographic primitives in the TLS 1.3 protocol, including ChaCha20-Poly1305, X25519, and Ed25519. Then, we describe the upper layer design of AsyncGBP in detail. Furthermore, a porting guideline is provided for vendors to apply the scheme to existing applications.

4.1 Cryptographic Primitives Optimization

We present efficient CUDA implementations of ChaCha20-Poly1305, X25519, and Ed25519 for the TLS 1.3 protocol, optimized for instruction execution, memory operations, and parallel mode.

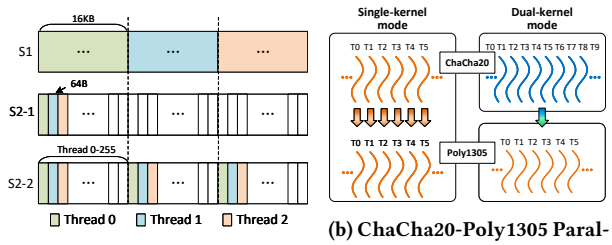
4.1.1 GPU-based ChaCha20-Poly1305. ChaCha20-Poly1305 is an AEAD algorithm that combines the stream cipher with the message authentication code (MAC). We optimize the two algorithms separately and discuss the parallel mode of the AEAD algorithm.

Instruction Optimization. In order to enhance the efficiency of time-consuming computations in the algorithm, we optimize the quarter-round function of ChaCha20 and the large integer multiplication in Poly1305 using PTX instructions. The quarter-round function involves ARX operations. Since there is no direct instruction for left rotation in CUDA PTX, we employ `shf` to implement it. To reduce the overhead introduced by handling multiplication overflow, we divide the 130-bit integers into five 26-bit limbs and optimize the multiplication operation leveraging instruction `mad.wide`.

Memory Accessing Optimization. ChaCha20-Poly1305 is a memory-intensive algorithm, where efficient and reasonable memory access is crucial for overall performance. To enhance the efficiency of data reading and writing, we utilize the built-in vector type [15] to optimize register access for ChaCha20 and limb conversion for Poly1305. In addition, we leverage `uint64` type to access data from global memory, improving the throughput of memory access.

The parallelism policy of ChaCha20 affects memory transaction efficiency. In coarse-grained parallelism, a single thread is responsible for processing a complete cryptographic task. As the size of the data block increases, the memory access intervals of this policy also increase, resulting in a decrease in memory access efficiency.

To address this issue, we propose a fine-grain parallelism policy for ChaCha20, shown as S2-1 and S2-2 in Figure 3a, where each GPU thread generates only a 64-byte keystream and concatenates the outputs of adjacent GPU threads to form the keystream required for the entire data block. The fine-grained parallelism policy maintains the same memory interval for each GPU thread, so the efficiency of memory access does not decline with the increase of data block size.



(a) ChaCha20 Parallelism Policy. (b) ChaCha20-Poly1305 Parallelism Policy.

Figure 3: Optimization for ChaCha20-Poly1305 Parallelism Policy.

Optimization of Parallel Mode. ChaCha20-Poly1305 can be viewed as the combination of stream cipher and MAC, where the parallelism of the former is higher than that of the latter. Therefore, the parallel mode has a significant impact on the overall performance. A straightforward approach, called single-kernel mode, implements both algorithms in the same kernel. In this mode, each GPU thread independently computes a ChaCha20-Poly1305 instance. However, as we previously stated in the analysis of the ChaCha20 parallelism policy, this approach results in a reduction of memory throughput as the data block size increases. Thus, we propose the dual-kernel mode as depicted in Figure 3b, where the two algorithms are implemented in separate kernels. This mode enables both kernels to adopt different parallelism strategies using different execution configurations. Specifically, the ChaCha20 kernel uses the fine-grained parallelism policy proposed earlier to overcome the shortcomings of the single-kernel mode, while the Poly1305 kernel still adopts the pattern of each thread computing one instance due to limitations in the algorithm structure.

4.1.2 GPU-based X25519 and Ed25519. 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 Reference [6].

Optimization for Finite Field Arithmetic. This method optimizes large integer addition/subtraction, multiplication (including multiplication-addition), and square operations by using PTX instructions, and proposes a method that only requires one round of carry-reduce.

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

Optimization for Ed25519. This method employs the extended twisted Edwards coordinates and pre-computation techniques to accelerate scalar multiplication. For fixed-point scalar multiplication, this method employs a portion-by-portion addition method and creates an offline pre-computation table in the form of $(x, y, x \times y, y-x, y+x)$ to accelerate the operation. For unknown point scalar multiplication, the offline pre-computation table approach is no longer suitable as the point is indefinite. Therefore, this method utilizes a fixed-window method and generates online pre-computation tables for each instance of unknown point scalar multiplication.

In addition, we optimize the usability of these cryptographic primitives to be suitable for practical applications. Specifically, we reduce the data transmission overhead between the host and the device through streaming technology and page-locked memory. Furthermore, we conduct experiments to determine the maximum number of cryptographic requests for a single transmission to enhance performance.

4.2 Overall Architecture Optimization

Optimized cryptographic primitives serve as the foundation for realizing high-performance heterogeneous accelerators. For optimal performance, targeted optimizations are necessary for each layer of the architecture. The details of AsyncGBP are shown in Figure 4.

4.2.1 Provider Compatibility Layer. One of the design goals of AsyncGBP is to maintain compatibility, which includes supporting the OpenSSL context and the provider mechanism. Furthermore, the asynchronous mode of OpenSSL provides the possibility to fully leverage the computing power of GPUs, where the process of sending cryptographic requests and retrieving results is asynchronous. Consequently, a mechanism is needed to notify the caller when the computation result is available.

Asynchronous Event Manager. The Asynchronous Event Manager (AEM) is specifically designed to handle asynchronous events, which forms 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, the AEM suspends the asynchronous job and returns control to the caller.

Typically, the handle can be either a file descriptor or a callback. File descriptors rely on system calls for their read and write operations, leading to frequent transitions between kernel and user mode. In high-concurrency environments, this pattern can result in significant performance overhead. As a result, we employ callbacks as the asynchronous notification handle.

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

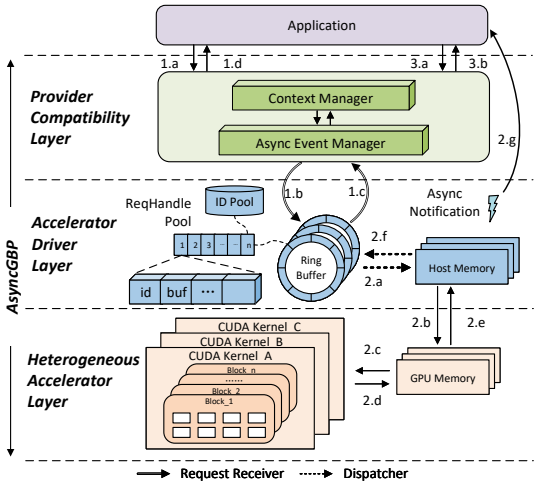


Figure 4: The Overall Architecture of AsyncGBP. (The steps in the figure correspond to the three steps in Figure 2 as follows: 1.a-1.d represents the process of request sending, 2.a-2.g describes the cryptographic computation process, and 3.a-3.b illustrates the process of result retrieval.)

Other Infrastructures. Additionally, we integrate the necessary functionalities into *ProvComp* to be compatible with the OpenSSL provider mechanism, such as provider initialization and teardown, cryptographic algorithm implementation queries. Considering that the HKDF [12] algorithm for TLS 1.3 cipher suite is not implemented by AsyncGBP, we adopt a hybrid policy to maximize the availability of the AsyncGBP. Specifically, for the cryptographic algorithms supported by the AsyncGBP, the implementations are set to high priority and provided to OpenSSL `libcrypto`; Otherwise, we mark them as low priority and reroute cryptographic requests to the default implementations of OpenSSL.

4.2.2 Accelerator Driver Layer. In AsyncGBP, GPUs serve as the final handler of cryptographic requests with a working mode different from OpenSSL. Therefore, we adopt the following measures to bridge the gap between these two working modes. Firstly, a considerable number of requests are cached before batch processing to fully leverage the parallel computing power of GPUs. Secondly, an efficient scheduling algorithm ensures high throughput of cryptographic computations while reducing latency as much as possible. To realize the above functions, we integrate two types of threads and three components in the Accelerator Driver Layer. Additionally, both the cryptographic request collection and GPU kernel scheduling threads are designed as multithreaded to decrease latency.

Lightweight Request Caching. Directly copying the data required for cryptographic operations to the buffer to collect requests is clearly inappropriate, as it results in redundant data copying and reduces the efficiency of request collection. To address this issue, we implement a lightweight request caching method. Specifically, our proposed method assigns a unique identifier for each cryptographic request. This identifier is utilized to retrieve a corresponding request instance and links it to the data associated with cryptographic computation. In this manner, we establish a chained relationship between the ⟨*ID, ReqHandle, Request*⟩ triad. By collecting identifiers, we achieve efficient and lightweight request aggregation, while eliminating redundant data copying.

The above process is implemented by the Request Receiver thread. In order to mitigate the overhead caused by temporary memory allocation, we establish an ID pool and a request instance pool, both of which support dynamic expansion as needed. The identifiers are stored in different ring buffers based on cryptographic request type, which leverages a strategy of space-time trade-off to mitigate the cost of sorting different types of requests. Given that two types of threads are bridged via the ring buffers, 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 conducting numerous experiments, we configure the initial sizes of the ID pool and the request instance pool as 8192, with the ring buffer size as 4096.

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

Concurrently accessing the same ring buffer by different threads in the multi-threaded system design causes conflicts and unnecessary waiting. In order to solve the issue, each dispatcher is assigned an initial offset that enables them to access different ring buffers at the same time, which is a small trick but significantly reduces the waiting time. After calling the GPU handlers to perform cryptographic computations, dispatchers also need to notify the application that the computation result is ready through asynchronous events. Due to the chained relationship of ⟨*ID, ReqHandle, Request*⟩, the real computation result is already in the memory allocated by the application, so the dispatcher can immediately recycle the identifier and request handle instance for the next use. Generating too many threads not only consumes CPU resources, but too many dispatchers also intensify the data race of ring buffers, resulting in

inefficient GPU utilization. After trial and error, we determine that the number of request receivers and dispatchers is both 4.

4.2.3 Heterogeneous Accelerator Layer. The Heterogeneous Accelerator Layer is mainly focused on implementing high-performance cryptographic algorithms. The collected cryptographic requests are forwarded to *HeteAcc* for computation under the control of the dispatcher. We provide CUDA-based handlers for the cryptographic algorithms required by TLS 1.3, including X25519 key exchange, Ed25519 signature generation and verification, and ChaCha20-Poly1305 encryption and decryption. In addition, We optimize the configuration based on hardware resource information to maximize the performance of each handler.

Asynchronous Data Transmission. 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. The data relating to cryptographic computation is sporadic in memory, which is not conducive to data transfer in bulk. As a solution, we need to copy the data into page-locked memory in the host and then utilize the asynchronous transfer mechanism 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 at a time. After repeated trials, each GPU handler is assigned a stream capable of transmitting up to 2048 cryptographic requests.

Scalability of Cryptographic Primitives. Given that the scalability of the underlying cryptographic algorithm is one of our design goals, we standardize the interface of the GPU processing programs. Specifically, We encapsulate operations such as data copying and GPU kernel invocation into a function whose entry parameter is an array of request identifiers. Furthermore, the initialization of GPU resources is structured for convenient extension of subsequent algorithms. The aforementioned approach decouples the caller from the implementation of underlying implementations, thereby enabling the easy modification and extension of cryptographic algorithms like post-quantum cryptography (PQC) in the future.

4.3 Deployment to Applications

The application architecture with integrated AsyncGBP is illustrated in the Figure 5. The left side of the diagram depicts the application, where the gray region indicates the original application and the green area represents the modifications for application vendors to implement. A simplified architecture of AsyncGBP is displayed on the right.

Hardware Environment. To meet the hardware prerequisites for implementing AsyncGBP, application vendors are required to equip their servers with GPUs.

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`. Furthermore, OpenSSL 3.0 or higher is a prerequisite for leveraging the provider mechanism. Finally, in order to enable GPU universal computing, it is necessary to install the GPU driver and CUDA toolkit.

Application Adaptation. In terms of programming, 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 correctly handle 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.

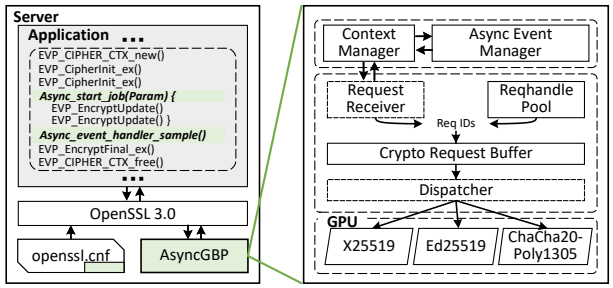


Figure 5: Deployment to Applications

By implementing the modifications mentioned above, application vendors can utilize AsyncGBP to offload the TLS 1.3 workload. Compared to rebuilding the entire application system, our proposed method is more practical and incurs lower migration overhead.

5 EVALUATION

In this section, we conduct a series of experiments to validate the effectiveness of the proposed cryptographic primitive optimization scheme. Furthermore, to evaluate the performance improvement of our design in TLS offload, we test AsyncGBP with a real-world application and compare its performance with other cryptographic acceleration schemes. The specific configurations in the experiment are detailed in Table 1.

Table 1: Experimental Platform Configuration.

CPU	Intel Xeon Gold 5218R @2.10 GHz
RAM	32GB
GPU	RTX 3070
OS	Ubuntu 20.04
Software	NVIDIA driver 510.108.03, CUDA 11.6, OpenSSL 3.0.0

5.1 Performance Evaluation of the Cryptographic Primitives

When evaluating the performance of cryptographic primitives, more attention should be paid to the *throughput* and *latency*. The *throughput* refers to the number of cryptographic operations completed within a unit of time, while *latency* represents the time interval between sending a cryptographic request and getting the computation result.

5.1.1 ChaCha20-Poly1305. Table 2 shows the kernel throughput of ChaCha20, Poly1305, and the combined AEAD mode, demonstrating the effectiveness of the proposed optimization schemes. The size of each data block is 16 KB, which is equal to the maximum size of each TLS record. The proposed optimizations all significantly enhance the throughput. When compared to the original version,

Table 2: Kernel Throughput Experiments of ChaCha20-Poly1305 on RTX3070.

	Batch Size	64	256	1024	4096	16394	65536
ChaCha20 (Gbps)	Coarse-grained Parallelism	18.64	60.48	235.6	741.04	578.96	490.16
	Fine-grained Parallelism	579.76	936.64	1333.76	1463.44	1499.28	**1508.16**
Poly1305 (Gbps)	Baseline	11.04	44	175.52	526.64	612.32	612.56
	Optimization	20.24	72.48	287.12	897.2	1052.24	**1146.08**
ChaCha20-Poly1305 (Gbps)	Single Kernel	8.64	31.28	122	417.12	377.12	354.32
	Dual Kernel	21.76	67.76	237.68	611.92	614.32	**672.64**

the optimized methods achieve peak performances that are 3.08x, 1.87x, and 1.90x higher, respectively.

Furthermore, the individual optimization methods for GPU-based ChaCha20 are evaluated as shown in Figure 6. It is worth noting that since each optimization method affects the consumed GPU resources, the kernel configuration needs to be tuned. The legend represents the superposition of all previous optimization methods. In conclusion, each optimization method makes its contribution to the throughput. Compared with the baseline, our scheme improves the throughput substantially, which is more significant when encrypting large data blocks. For example, the encryption throughput of 1 GB data blocks is 9 times that of the baseline.

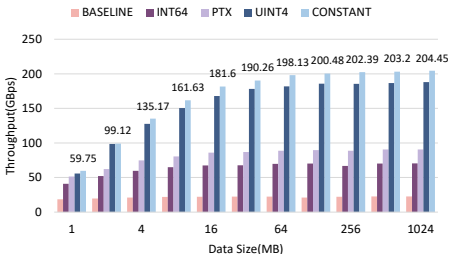


Figure 6: Impact of different optimizations on ChaCha20 kernel throughput.

5.1.2 X25519 and Ed25519. The performance of the cryptographic primitives is constrained by several factors, such as batch size and the number of available registers, which vary according to the hardware platform. The NVIDIA RTX 3070 features 46 Streaming Multiprocessors (SMs). In this architecture, each thread block can utilize a maximum of 65536 32-bit registers, whereas each thread can use up to 255 32-bit registers. After trial and error, we determine the optimal amount of registers for each thread, as shown in Table 3.

Table 3: Kernel Throughput and Latency Experiments of X25519 and Ed25519 on RTX3070.*

Threads/Block		128	256	384	512	640	768	896
Registers/Thread		255	255	168	128	96	80	72
X25519	(kops)	7711.44	10757.51	11281.43	**11728.68**	11443.01	11195.13	10961.52
	(ms)	0.76	1.10	1.57	1.92	2.57	3.16	3.76
Ed25519 SigGen	(kops)	11639.68	21686.60	26210.83	32472.98	**35739.26**	27706.67	24006.11
	(ms)	0.51	0.54	0.67	0.73	0.82	1.28	1.72
Ed25519 SigVer	(kops)	1991.39	2428.27	3755.89	4167.33	**4438.55**	3946.25	3423.33
	(ms)	2.96	4.85	4.70	5.65	6.63	8.95	12.04

*The number of blocks is fixed at 46, which is equal to the number of Streaming Multiprocessors on the RTX 3070.

Table 3 illustrates the throughput of X25519 scalar multiplication and Ed25519 signature generation and verification, where the number of threads per block is fixed at 46 (corresponding to the number of SMs in RTX 3070). The results represent that larger batch size benefits higher throughput, while the increase in latency remians acceptable. However, if the resource requirements exceed the threshold, variables spill into the local memory of GPU and performance begins to degrade.

5.2 Performance Evaluation of AsyncGBP

This subsection presents a performance evaluation of AsyncGBP. To ensure accurate and reliable results, we measure its performance using the command-line utility `openssl speed`. This utility leverages the EVP APIs to execute a specified cryptographic algorithm repeatedly within a predetermined time frame, then calculates the completed operations and reports the throughput.

Table 4: Performance Comparison in Multi-process Setting

		X25519 (kops)	Ed25519_SigGen (kops)	Ed25519_SigVer (kops)	ChaCha20-Poly1305 (GBps)
Local Test		9121.56	12398.76	4086.43	73.76
AsyncGBP	1 Proc.	1143.20	1058.74	1052.88	40.48
	2 Proc.	1873.46	1955.27	1843.12	**46.50**
	4 Proc.	3283.40	3175.38	2782.03	46.30
	6 Proc.	4509.69	5339.54	3432.10	41.62
	8 Proc.	**5573.11**	**6333.08**	**3967.72**	41.65

* "Local Test" describes the testing process that involves data transfer and kernel execution.

Previous analyses indicate that the callback-based asynchronous notification mechanism outperforms that based on file descriptors in high concurrency scenarios. Consequently, we apply a patch to the `openssl speed` to enable the callback-based asynchronous event notification mechanism that is not supported by default.

Table 4 presents the performance evaluation of AsyncGBP in a multi-process environment. In order to fully squeeze the computing power of the GPU, we enable the NVIDIA Multiple Process Service (MPS) for concurrent processing of the CUDA kernel on the same GPU. As indicated by the results, the performance of X25519 and Ed25519 improves progressively with an increase in the number of processes. The multi-process setting does not greatly enhance the performance of ChaCha20-Poly1305 due to frequent and substantial data copying between host and device. In comparison to the local test of cryptographic primitives involving kernel operations and data copying, AsyncGBP achieves 51%-97% of its performance. It can be inferred that the performance bottleneck of AsyncGBP is caused by the CPU or memory. Employing a higher-performance CPU would improve the throughput of AsyncGBP.

5.3 Related Work Comparison

This section presents a comparison between our work and implementations on CPU, GPU and FPGA. Furthermore, we conduct a performance evaluation of AsyncGBP compared to other cryptographic hardware accelerators.

5.3.1 vs. ChaCha20-Poly1305 Implementation. Wang et al. [27] accelerated ChaCha20 by coalesced memory access and branching operations and finally obtained a kernel throughput of 119.32 GBps on RTX 3070. Our scheme achieves a kernel throughput of 204.45

GBps on the same GPU, representing a 71.3% improvement compared to their scheme. Serrano et al. [24] reported a throughput of 0.948 Gbps for the ChaCha20-Poly1305 implemented on FPGA, while our optimized implementation achieved a peak throughput on NVIDIA RTX 3070 of 46.50 Gbps, which is two orders of magnitude greater than their performance.

5.3.2 vs. Other OpenSSL-compatible Cryptographic Accelerators. Table 5 presents a performance comparison between AsyncGBP and other OpenSSL-compatible cryptographic accelerators. AsyncGBP demonstrates significantly higher performance than the OpenSSL default provider. Intel QuickAssist Technology (QAT) is an ASIC-based hardware acceleration solution that supports multiple cryptographic primitives. Reference [10] evaluated the performance of QAT using `openssl speed`, where the throughput of X25519 is 120 kops. In comparison, our implementation of X25519 presents significantly higher performance, outperforming Intel QAT by an order of magnitude. Reference [23] described a hardware accelerator for TLS handshake based on FPGA, which can offload computationally intensive public key operations. Its ECDSA-P256 Signature performance is 910 kops, while the throughput of our proposed scheme outperforms this work by 7.0 times.

Table 5: Performance Comparison with Other OpenSSL-compatible Accelerators.

	X25519 (kops)	Ed25519_SigGen (kops)	Ed25519_SigVer (kops)	ChaCha20-Poly1305 (Gbps)
OpenSSL Default Provider [17]*	23.90	24.32	7.64	15.6
Intel QAT [10]	120	-	-	-
Secure-IC SCZ_SP_BA452 [23]†	1000	910	750	-
Ours	**5573.11**	**6333.08**	**3967.72**	**46.50**

*The experimental data of the OpenSSL default provider is obtained in a single-process setting.
†The performance of NIST Curve P-256.

6 CONCLUSION

In this paper, we proposed AsyncGBP, a heterogeneous and high-performance TLS acceleration framework, bridging the gap between the OpenSSL Provider mechanism and GPUs. The comprehensive performance evaluation demonstrated that a prototype of AsyncGBP outperforms other cryptographic accelerators by a wide margin. AsyncGBP highlights the potential of GPU-based cryptographic accelerators to enhance the throughput and efficiency of SSL/TLS in high-traffic environments. In addition, a practical implementation guideline is provided for application vendors to update their systems. Our future work includes adding support for post-quantum cryptography and further evaluating the performance in large-scale TLS concurrency scenarios.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under Grant No.2022YFB3103301 and National Natural Science Foundation of China under Award No.61902392.

REFERENCES

[1] Armin Ahmadzadeh, Omid Hajihassani, and Saeid Gorgin. 2018. A High-Performance and Energy-Efficient Exhaustive Key Search Approach via GPU on DES-like Cryptosystems. *The Journal of Supercomputing* 74, 1 (Jan. 2018), 160–182.
[2] Daniel J. Bernstein. 2005. The Poly1305-AES Message-Authentication Code. In *Fast Software Encryption (Lecture Notes in Computer Science)*, Henri Gilbert and Helena Handschuh (Eds.). Springer, Berlin, Heidelberg, 32–49.
[3] Daniel J Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography-PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings 9*. Springer, New York, 207–228.
[4] Daniel J Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. 2008. Twisted edwards curves. *Lecture Notes in Computer Science* 5023 (2008), 389–405.
[5] William N Chelton and Mohammed Benaissa. 2008. Fast elliptic curve cryptography on FPGA. *IEEE transactions on very large scale integration (VLSI) systems* 16, 2 (2008), 198–205.
[6] Jiankuo Dong, Fangyu Zheng, Jingqiang Lin, Zhe Liu, Fu Xiao, and Guang Fan. 2022. EC-ECC: Accelerating elliptic curve cryptography for edge computing on embedded GPU TX2. *ACM Transactions on Embedded Computing Systems (TECS)* 21, 2 (2022), 1–25.
[7] Harold Edwards. 2007. A normal form for elliptic curves. *Bulletin of the American mathematical society* 44, 3 (2007), 393–422.
[8] 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, Portland, 494–504.
[9] Omid Hajihassani, Saleh Khalaj Monfared, Seyed Hossein Khasteh, and Saeid Gorgin. 2019. Fast AES Implementation: A High-Throughput Bitsliced Approach. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2211–2222.
[10] Intel. 2022. Building Software Acceleration Features in the Intel Quick Assist Technology Engine for OpenSSL 1.1.1. Retrieved March 2, 2023 from https://www.intel.com/content/www/us/en/developer/articles/guide/building-software-acceleration-features-in-the-intel-qat-engine-for-openssl.html
[11] Simon Josefsson and Ilari Liusvaara. 2017. *Edwards-curve digital signature algorithm (EdDSA)*. Technical Report. Internet Research Task Force.
[12] Hugo Krawczyk and Pasi Eronen. 2010. *HMAC-based extract-and-expand key derivation function (HKDF)*. Technical Report. Internet Engineering Task Force.
[13] Adam Langley, Mike Hamburg, and Sean Turner. 2016. *Elliptic curves for security*. Technical Report. Internet Engineering Task Force.
[14] Peter L Montgomery. 1987. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation* 48, 177 (1987), 243–264.
[15] NVIDIA. 2023. Built-in Vector Types. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#built-in-vector-types.
[16] OpenSSL. 2023. ASYNC_start_job. https://www.openssl.org/docs/manmaster/man3/ASYNC_start_job.html.
[17] OpenSSL. 2023. OpenSSL. https://www.openssl.org/.
[18] OpenSSL. 2023. OpenSSL 3.0.0 Design. https://www.openssl.org/docs/OpenSSL300Design.html.
[19] SSL Qualys. 2023. Labs-SSL Pulse. https://www.ssllabs.com/ssl-pulse/.
[20] Eric Rescorla. 2018. *The transport layer security (TLS) protocol version 1.3*. Technical Report. Internet Engineering Task Force.
[21] Ousmane Sadio, Ibrahima Ngom, and Claude Lishou. 2019. Lightweight security scheme for mqtt/mqtt-sn protocol. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, Granada.
[22] Daniel AF Saraiva, Valderi Reis Quietinho Leithardt, Diandre de Paula, Andre Sales Mendes, Gabriel Villarrubia González, and Paul Crocker. 2019. Prisec: Comparison of symmetric key algorithms for iot devices. *Sensors* 19, 19 (2019).
[23] Secure-IC. 2022. TLS Handshake Hardware Accelerator. Retrieved March 2, 2023 from https://www.secure-ic.com/wp-content/uploads/2022/11/SCZ_SP_BA452_TLS_Handshake_Hardware_Accelerator_Product_Sheet_Web.pdf
[24] Ronaldo Serrano, Ckristian Duran, Marco Sarmiento, Cong-Kha Pham, and Trong-Thuc Hoang. 2022. ChaCha20–Poly1305 Authenticated Encryption with Additional Data for Transport Layer Security 1.3. *Cryptography* 6, 2 (2022), 30.
[25] Cihangir Tezcan. 2022. Key Lengths Revisited: GPU-based Brute Force Cryptanalysis of DES, 3DES, and PRESENT. *Journal of Systems Architecture* 124 (March 2022), 102402. https://doi.org/10.1016/j.sysarc.2022.102402
[26] Lipeng Wan, Fangyu Zheng, Guang Fan, Rong Wei, Lili Gao, Yuewu Wang, Jingqiang Lin, and Jiankuo Dong. 2022. A Novel High-Performance Implementation of CRYSTALS-Kyber with AI Accelerator. In *European Symposium on Research in Computer Security*. Springer, Copenhagen, 514–534.
[27] Ziheng Wang, Heng Chen, and Weiling Cai. 2021. A hybrid CPU/GPU scheme for optimizing chacha20 stream cipher. In *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/-SocialCom/SustainCom)*. IEEE, New York, 1171–1178.
[28] 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*. ACM, New York, 76–90.