An Efficient Elliptic Curve Cryptography Signature Server With GPU Acceleration

Wuqiong Pan, Fangyu Zheng, Yuan Zhao, Wen-Tao Zhu, *Senior Member, IEEE*, and Jiwu Jing, *Member, IEEE*

***Abstract*— Over the Internet, digital signature has been an indispensable approach to securing e-commerce and other online transactions requiring authentication. Concerning the computing costs of signature generation and verification, it has become a more and more common practice for security practitioners to outsource such computations from heavily loaded application servers called tenants to dedicated proxies like signature servers in the enterprise private cloud. In this paper, we present our high-performance signature server called Guess. It implements the elliptic curve digital signature algorithm (ECDSA) with 256-b key size on a Linux-powered commodity computer, harnessing a desktop graphics processing unit as a featured cryptographic accelerator. We demonstrate our experience in maximizing the computing power of Guess and also its capability to deliver such power to the tenants, which includes down-to-earth customization and optimization considering various hardware and software factors. Our comprehensive implementation of ECDSA is tested against intensive network traffic. Field experiments show that Guess achieves $T_s = 8.71 \times 10^6$ operations per second (OPS) for signature generation or $T_v = 9.29 \times 10^5$ OPS for verification, which is significantly faster than existent prototypes and products. Guess is a universal server that readily supports various categories of elliptic curve cryptographic schemes, such as digital signature, key agreement, and encryption.**

***Index Terms*— Security as a service, digital signature, elliptic curve cryptography (ECC), graphics processing unit (GPU), implementation of cryptography.**

I. Introduction

THE proliferation of the Internet has given rise to electronic commerce and other online transactions like digital publishing and software distribution, where security is a necessity. Digital signature is a fundamental approach to security, yet the involved *public key* operations are far more computation-expensive than symmetric operations (e.g., block/stream cipher and hash) and incur observable processing overhead.

Signature processing may become very demanding in high-throughput environments. For instance, Alipay, an online payment platform with the largest market share in China, set a record by processing up to 85,900 transactions per second during an online shopping festival on November 11, 2015 [1]. Assume Alipay merely adopts digital signature to secure the payments, and for each transaction it conducts 2 signature *verifications* (to verify the buyer's certificate and thus the identity, and his/her signature on the payment) and 1 signature *generation* (to ratify the transaction). Then, in merely 1 second Alipay needs to handle up to 85,900 signature generations and 171,800 verifications. This is a daunting task.

A. Signature as a Service

In this paper, following the business model of security as a service (SECaaS) [2], we focus on signature as a service (SIGaaS) as a case study, and present a tangible signature server called Guess. This is motivated by the observation that it has become a more and more common practice for security practitioners to outsource signature computations from application servers that are typically websites to dedicated proxies that are typically signature servers in the enterprise private cloud (EPC), so that the application servers as *tenants* can offload the computation costs of signature generation and/or verification to these specialized facilities. Again take Alipay as an example, which was started by Alibaba Group as a simple e-payment system and is now a website in its own right; it is possible that Alibaba adopts SIGaaS in its EPC to facilitate Alipay and Alibaba's other subsidiaries (e.g., Taobao and Tmall). Besides the common (and widely-recognized) benefits offered by SECaaS (and cloud computing in general), SIGaaS provides additional advantages in terms of security.

First of all, application servers like Alipay have to be openly accessible from the Internet and thus are potentially exposed to various attacks, some of which, like the Heartbleed exploit disclosed in 2014, may reveal websites' private keys, i.e., signing keys. With SIGaaS, the private keys are confined to standalone signature servers instead of being held by the public application servers. The input/output interfaces of a signature server are fairly simple: it accepts a message digest from and returns a generated signature to a tenant employing its private key, and accepts a digest-signature pair from and returns a boolean verification result to a tenant employing a specified public key. Therefore, the private key for signing (even the entire signature server) is beyond the "reach" of Internet attackers. This significantly mitigates the risk of key compromise.

Second, by shifting signing operations to signature servers, tenants are far less vulnerable to "stealthy" threats like

Manuscript received February 6, 2016; revised June 24, 2016; accepted July 19, 2016. Date of publication August 29, 2016; date of current version October 31, 2016. This work was supported in part by the National Natural Science Foundation of China under Grant 61272479, in part by the National 973 Program of China under Grant 2013CB338001, and in part by the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDA06010702. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Giuseppe Persiano. *(Corresponding author: Wen-Tao Zhu.)*

The authors are with Data Assurance and Communication Security Research Center (and also State Key Laboratory of Information Security, Institute of Information Engineering), Chinese Academy of Sciences, Beijing 100093, China (e-mail: wqpan@is.ac.cn; fyzheng@is.ac.cn; zhaoyuan12@is.ac.cn; wtzhu@ieee.org; jing@ieee.org).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2016.2603974

side-channel attacks. For example, it would be very difficult for an Internet attacker to launch timing attacks to infer a tenant's private key, as the actual signing is shifted to a back-end signature server, typically located in the EPC serving multiple tenants. According to our experimental data obtained from Guess, the service latency for signature generation is susceptible to factors unpredictable by outsiders like the overall traffic load on Guess, and our observation is in line with recent related work [3], [4]. Thus, the shifting to a back-end server introduces remarkable noise to the latency actually measurable by an Internet attacker, adding an extra layer of protection. (It is worth noting that Guess itself is designed to be resistant to timing attacks.)

B. Technical Challenges

Public key schemes like digital signature and key agreement are computation-expensive. Performing public key schemes with software may quickly saturate processors. For example, in one second, a modern central processing unit (CPU) core that is capable of processing over 10K HTTP requests can serve just more than 2K SSL transactions with RSA-1024, where the bottleneck lies in key agreement [5]. This also explains why busy application servers need to outsource public key computations to specialized facilities dedicated to such computations. Nevertheless, existent signature servers and similar equipments are still too slow to satisfy high-throughput tenants.

Assume Alipay adopts the elliptic curve digital signature algorithm (ECDSA). It was standardized by NIST in 2013 [6] in favor of its salient communication-efficiency compared with alternatives like RSA; for example, ECDSA with 256-bit key size ("ECC-256" for short in this paper) is comparable to RSA-3072 in security strength [7]. Suppose Alipay outsources the aforementioned task of 85,900 signature generations and 171,800 verifications to NShield Connect 6000+ [8], which is one of the fastest signature servers specifically optimized for elliptic curve cryptography (ECC) available on the market as of July 2014. According to [8], the server performs best with ECC-256, achieving 2400 operations per second (OPS) for signature generation, yet for some unknown reason its performance with signature verification is not specified. For ECDSA [6], signature verification is intrinsically more computation-expensive than generation, and thus the server should handle far less than 2400 OPS for verification. Therefore, Connect 6000+ needs more than $\frac{85,900}{2400} + \frac{171,800}{2400} = 107.375$ seconds for the signature computations. In other words, it may cost a signature server about 2 minutes to process the transactions taken place within merely 1 second on Alipay.

For a signature server (which is meant to fully take on the outsourced signature processing), it only makes sense when the computing task accumulated in a certain period can be accomplished within that period; otherwise, the transactions on the tenant's side cannot be completed in real time. Therefore, under the above estimation, Alipay has to deploy more than a hundred Connect 6000+ servers to just fulfill the workload. This is in sharp contrast to the SIGaaS vision that multiple tenants share the same signature server.

C. CPU, GPU, and GPGPU

A general-purpose CPU may have about two (or three) dozen cores that are optimized for sequential processing, and only handles a few threads per core. A large portion of the silicon area in a CPU is for a large cache hierarchy and sophisticated flow control like branch prediction and out-of-order execution [4], [5]. By contrast, a graphics processing unit (GPU), initially designed for computer graphics and image processing, devotes most of its die area to a large array of arithmetic logic units optimized for parallel processing. If used properly, GPUs can be more effective than CPUs for numerical computations.

GPUs' mass production for the gaming market has led to an attractive price-performance ratio, which results in increased interest in general-purpose computing on GPU (GPGPU) [9], i.e., introducing GPUs in computations traditionally handled by CPUs. This is materialized by GPU vendors like NVIDIA with GPU-accelerated computing, which uses a GPU together with a CPU to accelerate scientific and enterprise applications, where computation-intensive portions of an application are moved to the GPU while the remainder of the code still runs on the CPU. For developers, software extensions to programming languages are available. Particularly, compute unified device architecture (CUDA) is NVIDIA's parallel computing platform and programming model [10], [11], enabling C programmers to leverage GPGPU for various applications.

For the security community, the unprecedented computing power and relatively low cost of modern GPUs have made them also an ideal platform for both cryptography [3]–[5], [12]–[15] and cryptanalysis [16], [17]. Most research efforts adopt NVIDIA's products, though porting to AMD's may be applicable [5]. In this work, we use an EVGA video card [18] containing a desktop GPU released by NVIDIA in 2013.

A GPU's computing power is fully unleashed only when it is properly employed for parallel processing. Naive porting of cryptology algorithms to a GPU may cause severe performance degradation, wasting most of its computational resources. For example, a single GPU thread runs far slower than a CPU thread, and thus naive porting will yield unacceptable latency [5]. A GPU runs at full utilization only when a large number of equally-shaped tasks are processed simultaneously regarding respective inputs. The key to high-performance GPU-accelerated computing lies in customization.

D. Guess

In this paper, as our concrete implementation of SIGaaS, we present a GPU-accelerated universal elliptic-curve signature server, Guess for short. It is universal in that Guess supports various ECC (signature) schemes where scalar multiplication, also known as point multiplication (PM), is the chief cryptographic primitive. These includes ECDSA [6], EC-KCDSA [19, Sec. 4.4.2], the SM2 signature [20], and so on. EC-KCDSA is Korea's signature scheme, while SM2 is an ECC algorithm suite for digital signature, key agreement, and encryption, issued in late 2010 as the public key cryptographic standard in China (and included in late 2015 in ISO/IEC 14888-3). In fact, Guess readily supports the entire

SM2 suite. Among these, we choose ECDSA, the most widely standardized ECC signature scheme [19, Sec. 4.4.1], as the "target scheme" for Guess so that we can compare our work with a wide range of existent prototypes and products.

Guess is designed to be a standalone signature server whose capability is delivered via high-speed network to tenants typically located in a well-protected EPC (thus, in this paper we do not address issues like authentication between tenants and Guess, or transfer of private keys from tenants to Guess). Due to budget constraints, our platform is merely a combination of off-the-shelf commodity hardware and freely available software: it contains one 2.7GHz 12-core Intel Xeon E5-2697 v2 CPU [21], one NVIDIA GeForce GTX 780 Ti desktop GPU on a video card by EVGA [18], one 10Gb Ethernet controller, and 16GB DRAM; the OS is the 64-bit server edition of Ubuntu 12.04 and the compiler is from CUDA Toolkit 5.5. CUDA allows us to forthrightly boost Guess' performance by simply upgrading our fairly affordable platform with high-end hardware, e.g., substituting an 18-core CPU for our 12-core CPU, or replacing our GTX 780 Ti with a more modern GPU. In general, our C code can run on any Linux-powered computer after recompiling, possibly with a reconfiguration tailored to the new platform.

Two metrics are generally used for measuring the performance of a signature server. The first is *throughput*, i.e., how many transactions can be completed within a time unit. There are two types of ECDSA operations, *generation* (i.e., signing) and *verification* (i.e., verifying), and the throughput of the latter is far lower than that of the former due to ECDSA's imbalanced algorithms.[1] The second metric is the *latency* from a tenant's submission of service request to receiving the response (recall the server's I/O interfaces mentioned above). Note that high throughput and low latency may be conflicting goals [3], [4]; usually developers have to strike a balance between them. As a signature server for potential tenants like Alipay, Guess aims at high throughput without severely sacrificing latency. To this end, we need to customize and optimize our software implementation w.r.t. the given platform, so that we can maximize the platform's computing power as well as its capability to provide such power to the tenants.

A signature server is a complicated computing system, and its development and evaluation are a non-trivial engineering task. Before having to dive into the nuts and bolts of Guess, we sketch how we augment its throughput as follows. On a high level, the bottleneck of both types of ECDSA operations lies in PMs. Rather than adopting textbook implementations for PMs, we devise customized algorithms tailored to our device. On a low level, several memory spaces are available for CUDA programming [10, Sec. 9.2], among which on-chip *registers* are accessed with the lowest latency. In the low-level building blocks for PMs we rely on registers but optimize register usage, so that each GPU thread consumes a minimal number of registers. Tests show that on our platform, the overall latency for any ECDSA operation, even if processed by one thread, is fairly acceptable. Such a positive result implies that there is no need for us to parallelize any ECDSA operation *itself*. Then we simply launch N independent threads, each handling one transaction, so that the throughput is increased N times or so. We enlarge N till the hardware limit (when no more registers are available) to maximize the throughput.

An advantage of the above *one-thread-per-task* approach is that no computing resources are expended on synchronization or data exchange between associated threads that cooperate for one transaction (e.g., a signing/verifying operation). We save the resources so that Guess can launch in parallel as many as possible independent threads to unleash its full utility. This is in sharp contrast to the *multiple-threads-per-task* approach like the ones in [3] and [12] (where quite a few threads jointly compute a single PM to pursue low latency) and the one in our previous research effort [13] (where the RSA exponent is so large that one modular exponentiation has to be distributed to multiple collaborative threads). Essentially, which approach to adopt (in our case, the former) depends on the design goal and also the platform given. Interested readers can refer to [4] for some insightful discussions on the two approaches.

E. Technical Contributions and Paper Organization

Contributions of this paper are threefold. First, we propose an (almost exhaustive) implementation of ECDSA, customized and optimized for our GPU-accelerated computing platform to maximize its computing power. This effort translates ECC theory into down-to-earth realization on commodity processors. Second, to convey the computing power to network tenants with minimum loss, we combine the proposed algorithms with efficient resource allocation and scheduling. As a case study for SIGaaS, we develop a tangible signature server known as Guess based on a platform that is both fairly affordable and easily upgradable. Third, we conduct rigorous experiments to evaluate Guess and compare it with existent prototypes and products. Real performance data lead to the understanding that digital signature as a fundamental approach to Internet security, even if as strong as ECC-256, can be far more cost-effective than people used to conjecture (hence there seems to be no excuse for procrastinating adoption/upgrading).

The rest of this paper is organized as follows. Section II outlines technical backgrounds of ECC, ECDSA, and CUDA. Section III elaborates our full-featured implementation of ECDSA by detailing a wide range of algorithms tailed to our platform w.r.t. various hardware and software factors. Section IV specifies Guess' system architecture and resource management, which are independent of the target scheme and may be applicable to general signature server implementation. Section V presents performance data for evaluation and comparison. Finally, Section VI concludes this paper.

II. Preliminaries

This section begins with some basic theory of ECC. Then we move to ECDSA and explain why 256-bit key size is chosen for Guess. Last, we introduce necessary concepts for CUDA programming.

[1]The opposite holds for RSA: due to the specific structure of the public exponent (typically 65537), verifying is prominently faster than signing.

A. Elliptic Curve Cryptography (ECC)

Let $p > 3$ be a prime, $a, b \in \mathbb{F}_p$ satisfy $4a^3 + 27b^2 \neq 0$. An elliptic curve E over $\mathbb{F}_p$ is defined with the following equation:

$$y^2 = x^3 + ax + b, \quad (x, y) \in \mathbb{F}_p^2. \tag{1}$$

The addition operation on the curve, following the chord-and-tangent rule [19, Sec. 3.1.2], is called *point doubling* when the two points are identical, or *point addition* otherwise. All the points on the curve and the point at infinity ∞ form an additive Abelian group $E(\mathbb{F}_p)$, with $\infty(= -\infty)$ serving as the identity element. NIST adopts the selection of $a = -3$ in (1) for efficiency reasons, and recommends 5 curves [6, App. D.1.2] where fast reduction over $\mathbb{F}_p$ is applicable [19, Sec. 2.2.6].

The coordinate system in (1) is called the affine system, where point addition involves an expensive inversion. This can be worked around if we trade storage for computation and switch to the Jacobian system, where a point $(x, y) \in E(\mathbb{F}_p)$ is represented with 3 coordinates $(X : Y : Z) \in \mathbb{F}_p^3$ satisfying $x = \frac{X}{Z^2}, y = \frac{Y}{Z^3}$. For example, ∞ corresponds to $(1 : 1 : 0)$, and the negative of $(X : Y : Z)$ is $(X : -Y : Z)$ [19, Sec. 3.2.1]. Low-level algorithms for point addition/doubling in the Jacobian system are summarized in [15, Appendix A.1].

Point multiplication (PM) is defined as multiple point additions: for $k \in \mathbb{Z}^+$, $P \in E(\mathbb{F}_p)$, kP is the sum of k P's. PM dominates the execution time of ECC schemes [19, Sec. 3.3].

B. Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA is the most widely standardized ECC signature scheme, appearing in ANSI, FIPS, IEEE, IETF, ISO/IEC, and other standards. Let G be the base point with a prime order n on the curve, where n is very close to the p in (1) as per Hasse's Theorem [19, Sec. 3.1.3]. Then, $|n|$ is called the *key size*, and ECDSA produces digital signature (r, s) of twice the length of the key size, i.e., $2|n|$. Legacy standards specified a minimum key size of 160 bits (i.e., ECC-160), which is no longer recommended after 2013; for 2014 through 2030, signing with ECC-224 is still acceptable, yet for 2031 and beyond, at least ECC-256 (corresponding to 128-bit security) has to be adopted [7, Sec. 5.6.1,5.6.2]. (However, those who are conservative may be reluctant to upgrade to stronger security; the cost of new facilities may be an excuse.) Commercial products like Luna SA 7000 [22] and the aforementioned NShield Connect 6000+ [8] are also optimized for ECC-256. Hence, we select ECDSA with key size $|n| = 256$ bits as the "target scheme" for Guess, and accordingly adopt the Curve P-256 [6, Appendix D.1.2.3] recommended by NIST.

In ECDSA, the private key for signing is $d \in_R \mathbb{Z}_n^*$ and the public key for verifying is $Q = dG$. The signature generation and verification [19, Sec. 4.4.1] are shown in Algs. 1 & 2, where $e = h(m)$ is the digest of the input message m, generated by a tenant with a secure one-way hash function $h(*)$ satisfying $|e| \geq |n|$ [6, Sec. 6.1.1] (a hash is a very cost-efficient operation and tenants do not need to outsource it; for $|n| = 256$, $h(*)$ can be SHA-256 [23]). Digests rather than varying-size messages are sent from the tenants, so that the messages are transparent to the signature server (as it does not need to know them) and the throughput simply increases linearly with the traffic.

Algorithm 1 ECDSA Signature Generation Algorithm

Input: private key d, digest e **Output:** sig. (r, s)

1: select $k \in_R \mathbb{Z}_n^*$
2: $(x_1, y_1) = \underline{kG}$
3: $r = x_1 \bmod n$; go to step 1 if $r = 0$
4: $s = k^{-1}(e + dr) \bmod n$; go to step 1 if $s = 0$

Algorithm 2 ECDSA Signature Verification Algorithm

Input: public key Q, digest e, sig. (r, s) **Output:** accept/reject

1: reject unless $r, s \in \mathbb{Z}_n^*$
2: $w = s^{-1} \bmod n$
3: $u_1 = ew \bmod n, u_2 = rw \bmod n$
4: $(x_1, y_1) = \underline{u_1 G + u_2 Q}$; reject if $(x_1, y_1) = \infty$
5: accept if $r = x_1 \bmod n$, reject if not

Clearly, signature generation is faster than verification. The most computation-heavy phase of Alg. 1 is step 2 and that of Alg. 2 is step 4. Since the base point G is fixed for the Curve, kG in Alg. 1 and $u_1 G$ in Alg. 2 are called *fixed point* multiplications. In Alg. 2, as the input Q is not known a priori, $u_2 Q$ is called an *unknown point* multiplication [19, Sec. 3.3].

C. CUDA Programming

CUDA programming involves executing code in two distinct settings: a *host* (e.g., a computer) with (at least) one CPU, and a *device* (e.g., a video card) with (at least) one GPU. They differ in hardware and software aspects like memory space and threading model [10, Sec. 2.1]. In the view of heterogeneous programming [11, Sec. 2.4], serial code runs on the host while parallel code runs on the device, which operates as a coprocessor to the host. In CUDA programming, the *global memory*, which resides in the device DRAM and is the most plentiful among various device memory spaces, can be accessed and modified by both the host and the device.

A GPU is built around an array of streaming multiprocessors (SMs) [11, Sec. 4]. Ours, GeForce GTX 780 Ti, is of "compute capability" 3.x, for which an SM contains 192 cores [11, Appendix G.4]. It has 15 SMs, amounting to 2880 cores. A multithreaded program is partitioned into independent *blocks* of threads. An SM hosts multiple blocks, and a block can be scheduled on any available SM. Thus, a GPU with more SMs will automatically execute the program in less time than one with fewer SMs.

CUDA extends C by allowing the programmer to declare a function as a *kernel* [11, Sec. 2.1], which when invoked by the host runs in parallel by N threads on the device. That is, a kernel is instantiated N times (as opposed to only once for a regular C function). As mentioned above, these threads are organized as (in other words, a kernel is executed by) N_b equally-shaped blocks. Denote the number of threads per block (e.g., 256) as N_t. Then we have $N = N_b \times N_t$ [11, Sec. 2.2]. Both N_b and N_t are specified by

the programmer. Due to physical constraints on a GPU, there are limits to N_b, N_t, and N [11, Appendix G.1].

III. ECDSA Implementation

This section details our comprehensive implementation of ECDSA. From a developer's perspective, our technique can be decomposed into two levels (cf. Section I-D): the high-level algorithms translate mathematics into general software descriptions, which are then supported by the low-level algorithms consisting of assembly-like instructions for the GPU. The high-level algorithms mainly address PMs, where we pursue solutions that are even more efficient than standard (or termed as textbook) accelerations. The low-level algorithms focus on memory optimizations [10, Sec. 9].

A. Pre-Computing Table Aided Fixed Point Multiplication

PM dominates the execution time of ECC schemes including ECDSA. Clearly, a tenant cares most about the service latency, but is blind to the ways Guess implements ECDSA operations. Our implementation choices are not necessarily limited to the many textbook approaches, but have to suit and well harness the given platform. Particularly, our device, the EVGA video card [18], has affluent (3GB) global memory, only a small part of which should be reserved for data exchange between the host and the device (cf. Section II-C). As a result, for *fixed point* multiplication, we choose to trade storage for computation, employing a pre-computing table (PCT) built offline w.r.t. the fixed point G.

Should Guess support different ECC signature schemes for different tenants, a PCT would have to be built for each curve. For ECDSA with Curve P-256, only one PCT is needed, which can be generated using any approach and saved in a data sheet only once. Upon booting, Guess loads the entire PCT into the device's global memory for readonly access later. The larger the PCT, the faster the fixed point multiplication, and our experiments show that a 64MB PCT is beneficial enough for acceptable latency. In theory, using an even larger PCT we may boost Guess' performance forthrightly, but we find that the bottleneck of Guess then lies in service delivery rather than PM or other cryptographic processing (thus employing an even larger PCT does not help much in itself).

Now let us be more specific. Take kG for example. Let $k = k_{l-1}||\dots||k_1||k_0$ be its radix-2^f form, i.e., $k = \sum_{i=0}^{l-1} 2^{if} k_i$, where $|k_i| = f$, $fl \geq |k| = |n|$. Since $kG = \sum_{i=0}^{l-1} 2^{if} k_i G$, the PCT needs to contain $2^{if} k_i G$ for all $0 \leq k_i \leq 2^f - 1, 0 \leq i \leq l-1$, altogether $2^f l$ items, with each item of size $|(x, y)| = 2|n|$ bits. Since $|n| = 256$, we can select $f = l = 16$, resulting in a PCT of size $2|n| \times 2^f l$ bits, which is 64MB. Now that each $2^{if} k_i G$ is immediately available via table lookup, the fixed point multiplication $kG = \sum_{i=0}^{l-1} 2^{if} k_i G$ is simply reduced to $l-1$ point additions. Such reduction also prevents the nonce k (and thus the private key d) from being leaked to a side-channel attacker [24], [25] thanks to the table lookups.

As mentioned earlier, we address the implementation of ECDSA on Guess from two levels. Next, we switch from the fixed point multiplication on the high level to point additions on the low level. We begin with the general case in Section III-B, and then consider a special case in Section III-C that is actually invoked when computing $kG = \sum_{i=0}^{l-1} 2^{if} k_i G$.

TABLE I

Point Addition in the Jacobian System, Where $P_i = (X_i{:}Y_i{:}Z_i) \in \mathbb{F}_p^3$. Input: P_1, $P_2 (\neq \pm P_1)$. Output: $P_3 = P_1 + P_2$

Step (calculation over $\mathbb{F}_p$)	Variables that need saving
load P_1, P_2	$X_1, Y_1, Z_1, X_2, Y_2, Z_2$
$R_1 = X_1 Z_2^2$	$R_1, Y_1, Z_1, X_2, Y_2, Z_2$
$R_2 = X_2 Z_1^2$	$R_1, R_2, Y_1, Z_1, Y_2, Z_2$
$R_3 = R_2 - R_1$	$R_1, R_3, Y_1, Z_1, Y_2, Z_2$
$R_4 = Y_1 Z_2^3$	$R_1, R_3, R_4, Z_1, Y_2, Z_2$
$R_5 = Y_2 Z_1^3$	$R_1, R_3, R_4, R_5, Z_1, Z_2$
$Z_3 = Z_1 Z_2 R_3$	R_1, R_3, R_4, R_5, Z_3
$R_6 = R_5 - R_4$	R_1, R_3, R_4, R_6, Z_3
$R_7 = R_3^2 R_1$	R_3, R_4, R_6, R_7, Z_3
$R_8 = R_3^3$	R_4, R_6, R_7, R_8, Z_3
$X_3 = R_6^2 - 2R_7 - R_8$	$R_4, R_6, R_7, R_8, X_3, Z_3$
$Y_3 = (R_7 - X_3)R_6 - R_4 R_8$	X_3, Y_3, Z_3

B. Optimized Jacobian Addition

To calculate point addition efficiently, we consider converting the chord-and-tangent rule [19, Sec. 3.1.2] into a low-level algorithm in the Jacobian system (cf. Section II-A). This involves certain formula deduction. For page limits, we omit the mathematics and describe in Table I our intermediate result, a "not-too-low-level" point addition working flow, the correctness of which can be easily checked against the chord-and-tangent rule. Note that point addition and point doubling (to be addressed later) are distinct; they follow different laws.

Table I lists the variables that need to be stored after each step and shows that our working flow is very *variable-efficient*: throughout the flow we never use more than 6 variables. This reaches the lower bound: with less than 6 variables, one cannot even load the two addends P_1 and P_2 at the very beginning. Note that at this stage we do not count the auxiliary variables used within a certain step; we will consider them shortly.

For GPU programming, memory optimizations are the most important area for performance. The general guideline is using as much fast memory and as little slow memory as possible [10, Sec. 9], which explains why we rely so much on registers. Registers are accessed with the lowest latency but are a limited resource; our GPU, GeForce GTX 780 Ti, is of "compute capability" 3.5 and has only 64K 32-bit registers per SM [11, Appendix G.1]. In Table I, each variable is of $|n|$ bits in length and is accommodated by $\frac{256}{32} = 8$ registers (called *field registers* in [15]). We pursue *variable efficiency* as it contributes to high performance: reducing the variables involved in (low-level) algorithms means saving the registers consumed per thread, and thus more threads can be launched in parallel to increase Guess' throughput (cf. Section I-D).

In Section III-A, the fixed point multiplication has been reduced to $l-1$ point additions, but they involve an "imbalanced" scenario: the first addend P_1 can be kept in the registers (which can also be reused for the sum), but each pre-computed item $2^{if} k_i G$ as the second addend P_2 has to be read from the global memory. Although abundant enough to accommodate the PCT, the global memory has the greatest access latency among all memory spaces (registers are at the other end of

TABLE II

THE PROPOSED LOW-LEVEL POINT ADDITION SCHEME IN THE JACOBIAN SYSTEM, WHERE $P_i = (X_i{:}Y_i{:}Z_i)$. INPUT: P_1 IN REGISTERS, $P_2(\neq \pm P_1)$ IN MEMORY. OUTPUT: $P_1 = P_1 + P_2$ IN REGISTERS

(1) $T_1 = Z_1^2$, (2) $T_2 = T_1 Z_1$, (3) $T_1 = T_1 X_2$ $(= R_2)$,
(4) $T_2 = T_2 Y_2$ $(= R_5)$, (5) $T_3 = Z_2$, (6) $Y_1 = Y_1 T_3$, (7) $Z_1 = Z_1 T_3$,
(8) $T_3 = T_3^2$, (9) $Y_1 = Y_1 T_3$ $(= R_4)$, (10) $X_1 = X_1 T_3$ $(= R_1)$,
(11) $T_1 = T_1 - X_1$ $(= R_3)$, (12) $Z_1 = Z_1 T_1$ $(= Z_3)$,
(13) $T_2 = T_2 - Y_1$ $(= R_6)$, (14) $T_3 = T_1^2$, (15) $T_1 = T_1 T_3$ $(= R_8)$,
(16) $T_3 = T_3 X_1$ $(= R_7)$, (17) $X_1 = T_2^2$, (18) $X_1 = X_1 - T_3$,
(19) $X_1 = X_1 - T_3$, (20) $X_1 = X_1 - T_1$ $(= X_3)$, (21) $T_3 = T_3 - X_1$,
(22) $T_3 = T_3 T_2$, (23) $T_1 = T_1 Y_1$, (24) $Y_1 = T_3 - T_1$ $(= Y_3)$.

the spectrum). Hence, to change the "not-too-low-level" working flow in Table I into a low-level algorithm (i.e., a sequence of assembly-like instructions), we should take the imbalanced addends into consideration and minimize memory access.

Table II depicts our customized low-level point addition scheme, where each step is only an assembly-like instruction. Its correctness can be easily checked against Table I (the sequences of computing in the two tables differ slightly, which does not matter). Table II is compatible with existent algorithms like [15, Algorithm 15], but is more variable-efficient: only 6 variables ($X_1, Y_1, Z_1, T_1, T_2, T_3$) are needed to store the results throughout our proposal, while by contrast [15, Algorithm 15] uses 7 variables (and the same number of instructions). For both addition schemes, two auxiliary variables are needed and can be reused when computing any multiplication over $\mathbb{F}_p$; we do not need to count these two for comparison.

Concerning the aforementioned "imbalanced" scenario, Table II achieves minimal access to global memory. Specifically, X_2 and Y_2 are accessed only once, i.e., for the multiplications in steps (3) and (4), respectively; Z_2 is also accessed only once, i.e., for the assignment in step (5). The three coordinates are accessed one by one (instead of simultaneously) so that we can save the register usage. We also handle the multiplications attentively, as detailed in our previous work [14]. Take $T_1 = T_1 X_2$ in step (3) for instance. We employ the scanning algorithm [19, Sec. 2.2.2] for the multiplication, where we multiply T_1 by one (32-bit) word of X_2 at a time. That is, X_2 is read word by word, for which just one register (instead of field registers for the whole variable) is needed. Note that this paradigm [14] is inapplicable to squaring like X_2^2, where X_2 as either one of the operands has to be loaded entirely; we manage to work around such squaring in Table II.

This subsection might be a bit difficult to understand as it involves two kinds of computations: elliptic curve arithmetic (point multiplication/addition) and finite field arithmetic (integer multiplication/subtraction). A brief summary is as follows. Guess can employ Table II for imbalanced (or balanced) point addition, as the proposal is both variable-efficient in register utilization and optimal in memory access.

C. Optimized Mixed Jacobian-Affine Addition

The addition of two points in different coordinate systems is called *mixed* addition. We are interested in mixed addition of $(X : Y : Z)$ in Jacobian coordinates and (x, y) in affine coordinates, which can be simply implemented with an existent Jacobian addition algorithm by regarding (x, y) as $(x : y : 1)$ (cf. Section II-A). So, specifying $Z_2 = 1$ in Table II, we obtain Table III as our mixed addition algorithm, which is compatible with, yet one variable more efficient than [15, Algorithm 16].

TABLE III

THE PROPOSED LOW-LEVEL POINT ADDITION SCHEME IN THE MIXED JACOBIAN-AFFINE SYSTEM. INPUT: $P_1 = (X{:}Y{:}Z)$ IN REGISTERS, $P_2(\neq \pm P_1) = (x, y)$ IN MEMORY. OUTPUT: $P_1 = P_1 + P_2$ IN REGISTERS

(1) $T_1 = Z^2$, (2) $T_2 = T_1 Z$, (3) $T_1 = T_1 x$, (4) $T_2 = T_2 y$,
(5) $T_1 = T_1 - X$, (6) $Z = Z T_1$, (7) $T_2 = T_2 - Y$, (8) $T_3 = T_1^2$,
(9) $T_1 = T_1 T_3$, (10) $T_3 = T_3 X$, (11) $X = T_2^2$, (12) $X = X - T_3$,
(13) $X = X - T_3$, (14) $X = X - T_1$, (15) $T_3 = T_3 - X$,
(16) $T_3 = T_3 T_2$, (17) $T_1 = T_1 Y$, (18) $Y = T_3 - T_1$.

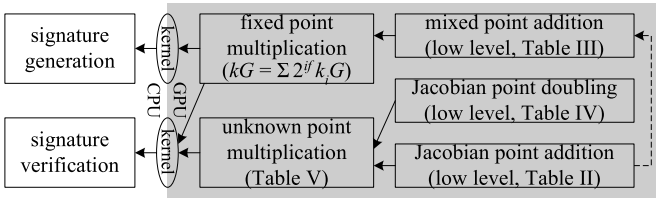

Fig. 1. Implementing ECDSA operations with ECC building blocks. An arrow from A to B means that A is a building block (bb) of B. Particularly, a low-level algorithm (in the right column) is a bb of a high-level algorithm (in the middle column), which in turn is a bb of a kernel. The dashed line indicates that the mixed point addition is a special form of the Jacobian point addition.

Actually, Guess invokes Table III (instead of Table II, as shown in Fig. 1) $l-1$ times for computing one PCT-accelerated fixed point multiplication, because the mixed Jacobian-affine addition as per Table III is more efficient than the Jacobian addition as per Table II. (The price is that the second addend P_2 needs to be saved in the affine system. Fortunately, the conversion is done during the offline generation of the PCT.) For variable efficiency, in the $l-1$ point additions, the output of a previous calculation (in registers, saving the sum initialized as $k_0 G$) is immediately used as an input of the next calculation.

So far we have only compared our low-level algorithms with the literature [15] in terms of space efficiency. Regarding fixed point multiplication, one may be concerned with the actual performance of our high-level implementation. In the literature, the most efficient algorithm for fixed point multiplication is the fixed-base comb method [19, Sec. 3.3.2], which also uses certain pre-computing but additionally involves point doublings. Compared with our implementation (which only invokes the low-level Table III $l-1$ times), the fixed-base comb method might exhibit less processing overhead when both schemes employ respective PCTs of roughly the same table size *TS*. However, the advantage comes along only with a very small *TS*, and declines when *TS* increases. When *TS* is large enough (e.g., 64MB), our proposal incurs significantly less computational overhead (e.g., about a half) than the fixed-base comb method. That is, for a device with sufficient memory like a video card, our implementation is much more efficient.

TABLE IV

THE PROPOSED LOW-LEVEL POINT DOUBLING SCHEME IN THE JACOBIAN SYSTEM, WHERE $P = (X{:}Y{:}Z)$. INPUT: P IN REGISTERS. OUTPUT: $P = 2P$ IN REGISTERS

(1) $T_1 = Z^2$, (2) $T_2 = X - T_1$, (3) $T_1 = X + T_1$, (4) $T_2 = T_2T_1$,
(5) $T_1 = T_2 + T_2$, (6) $T_1 = T_2 + T_1$, (7) $Y = Y + Y$, (8) $Z = YZ$,
(9) $Y = Y^2$, (10) $T_2 = YX$, (11) $Y = Y^2$, (12) $Y = Y/2$,
(13) $X = T_1^2$, (14) $X = X - T_2$, (15) $X = X - T_2$, (16) $T_2 = T_2 - X$,
(17) $T_1 = T_1T_2$, (18) $Y = T_1 - Y$.

TABLE V

THE PROPOSED UNKNOWN POINT MULTIPLICATION SCHEME. INPUT: $k = k_{l-1}||\ldots||k_1||k_0$ IN RADIX-2^f, $Q \in E(\mathbb{F}_p)$. OUTPUT: $S = kQ$

(1) pre-compute $(X{:}Y{:}Z)_i = iQ$, $i = 0, 1, \ldots, 2^f - 1$;
(2) $S = k_{l-1}Q$ (any k_iQ is fetched from the PCT); $i = l - 2$;
(3) $S = 2^fS$ (f Jacobian doublings following Table IV);
(4) $S = S + k_iQ$ (Jacobian addition following Table II);
(5) $i = i - 1$; if $i \geq 0$, go to (3).

D. Optimized Jacobian Doubling

Our point addition scheme proposed in Table II is not for cases of $P_1 = \pm P_2$, nor is its special form shown in Table III. For $P_1 = -P_2$, the sum is ∞. For $P_1 = P_2$,[2] we need an *independent* point doubling algorithm. To this end, as in Section III-B, again we begin with the chord-and-tangent rule [19, Sec. 3.1.2], deduce a "not-too-low-level" working flow like Table I (omitted here for space concerns), and eventually come to an optimized low-level algorithm depicted in Table IV.

E. Optimized Unknown Point Multiplication

In Section III-A we accelerate *fixed point* multiplication with a PCT built *offline*. Now we get back to the high-level design and address *unknown point* multiplication (u_2Q in Alg. 2) likewise, but with a different PCT. To begin with, the basic double-and-add algorithm [19, Sec. 3.3.1, Algorithm 3.27] for unknown point multiplication $S = kQ$ is as follows. Regard k as $(k_{|k|-1}, \ldots, k_1, k_0)_2$. Initialize the sum S with ∞ and a counter i with $|k| - 1$. Then, (i) let $S = 2S$; (ii) if $k_i = 1$, let $S = S + Q$; (iii) let $i = i - 1$; (iv) if $i \geq 0$, go to (i).

On Guess, we use a PCT-aided "window method" variant of the above algorithm. As Q is not known a priori, the PCT is built *on the fly* and thus has to be moderately small. Again let $k = k_{l-1}||\ldots||k_1||k_0$ be its radix-2^f form, where $|k_i| = f$, $fl \geq |k| = |n|$ (cf. Section III-A). We first compute iQ's for all $i < 2^f$ and then follow a similar double-and-add routine, as summarized in Table V.

For PCT-aided fixed point multiplication (cf. Section III-A), we take the trouble to convert each pre-computed item into the affine system so that we can employ the more efficient mixed Jacobian-affine addition; such conversion is done during the *offline* generation of the PCT and thus does not incur any online cost. Now, by contrast, for PCT-aided unknown point multiplication shown in Table V, we save each pre-computed item directly in the Jacobian system at step (1) and then simply employ Jacobian addition at step (4); otherwise, converting the PCT items *online* involves so expensive inversions that the conversion offsets the benefit of replacing Jacobian addition with the mixed Jacobian-affine addition. Such a difference between the fixed point and the unknown point multiplication implementations is also shown in Fig. 1.

As Jacobian addition (Table II) is less efficient than doubling (Table IV), a trick may be applied to step (1): compute iQ as the doubling of $\frac{i}{2}Q$ if i is even, as $(i-1)Q + Q$ if not.

In Table V, iQ's are pre-computed on the fly, which have to be *saved* in the global memory since the GPU does not have sufficient registers (or other memory space like so-called shared memory) to accommodate even a small PCT; then, the pre-computed k_iQ's are *fetched* from the global memory for every i. Access to the global memory (rather than point doublings and additions) complicates the performance of Table V, where it is difficult to theoretically infer the f that optimizes the algorithm's efficiency. Hence, we turn to an experimental approach and test every possible f. In the end, $f = 4$ turns out to be the optimal and thus is chosen (very surprisingly, we find that $f = 3$ is almost as good as $f = 4$, though $3 \nmid 256$). Note that when $f = 1$, the pre-computation is eliminated and Table V degenerates to the basic double-and-add algorithm.

F. Optimized Multiple Point Multiplication

In [19, Sec. 3.3], the combination $u_1G + u_2Q$ in Alg. 2 is called the *multiple point* multiplication, and is computed as a whole for potential speedup. In practice, however, we simply treat $u_1G + u_2Q$ as the sum of two separate parts, a fixed point multiplication and an unknown point multiplication. Since both parts have been implemented with optimized algorithms (cf. Fig. 1), it turns out that on Guess our simple divide-and-conquer approach actually achieves higher efficiency than the textbook one [19, Sec. 3.3.3].

G. Optimized (Explicit or Implicit) Modular Inversion

So far we have addressed PMs with efficient implementations. A review of Algs. 1 & 2 indicates that after optimizing PMs, modular inversions like $k^{-1} \bmod n$ then stick out as computation-heavy primitives. (In Section II-A the Jacobian system is introduced to avoid field inversions, but in Algs. 1 & 2 the inversions modulo n are explicit and unavoidable.) In less demanding applications, modular inversions might be considered lightweight. However, to build a signature server for intensive operations, we need to seriously address these primitives, which turn out to be a different story from PMs.

There are quite a few approaches to the modular inversion. Some like the Fermat's Little Theorem based one [19, Sec. 2.2.4] are straightforward but computation-prohibitive (involving an exponent as large as nearly 2^{256}).

[2]This is frequently the case in unknown point multiplication (to be addressed shortly), but impossible in PCT-accelerated fixed point multiplication. In Section III-A, $k = \sum_{i=0}^{l-1} 2^{if}k_i$, and thus a non-$\infty$ $\sum_{i=0}^{j-1} 2^{if}k_iG$ is not equal to $2^{jf}k_jG$ for $1 \leq j \leq l-1$ unless the condition $\sum_{i=0}^{j-1} 2^{if}k_i \equiv 2^{jf}k_j \pmod n$ holds for some j. Since n is close to $2^{|n|} - 1$, the condition might only be possible when $j = l - 1$, i.e., $\sum_{i=0}^{l-2} 2^{if}k_i + n = 2^{f(l-1)}k_{l-1}$. In our case, $|n| = 256$, $f = l = 16$, and the largest possible k_{15} is $2^f - 1$, but the n specified in Curve P-256 [6, Appendix D.1.2.3] is larger than $2^{32\times7}(2^{32} - 1)$, which is already larger than $2^{16\times15}(2^{16} - 1)$. So, the condition never holds.

Others like the extended Euclidean one and the binary one [19, Sec. 2.2.5] involve condition checks in iterations, which are more applicable to CPUs rather than GPUs (cf. Section I-C). The reason is as follows. The threads in a GPU are organized into groups and can only execute equally-shaped flows; thread divergence can significantly impact the effective instruction throughput [11, Sec. 5.4.2]. For GPU programming, one needs to minimize or even avoid different execution paths [10, Sec. 12.1]. (Hence, in Section III-E we do not use textbook algorithms such as the sliding window method [19, Sec. 3.3.1]. Note that in Table V, the judgment at step (5) is for looping and never leads to real branching between the threads.)

In short, we identify that the GPU is not an ideal platform for modular inversion. Thus, we turn to the alternative platform, the CPU. We adopt an efficient algorithm known as the simultaneous inversion [19, Sec. 2.2.5]. It takes as input an arbitrary number N_e of nonzero elements and outputs their inverses modulo a fixed prime, which concerning the *explicit* inversions in Algs. 1 & 2 is the order n of Curve P-256. The algorithm in itself is serial. It is called simultaneous inversion because all N_e elements should be given simultaneously as a batch input (our SIGaaS scenario exactly satisfies this requirement). The algorithm trades storage for computation; inverting N_e elements only incurs one inversion and $3(N_e-1)$ modular multiplications, where the only inversion can be dealt with the binary algorithm [19, Sec. 2.2.5]. For a large N_e, computing one inverse basically corresponds to 3 modular multiplications.

Converting a point $(X{:}Y{:}Z) \in \mathbb{F}_p^3$ in the Jacobian system into $(x = \frac{X}{Z^2}, y = \frac{Y}{Z^3})$ in the affine system *implicitly* involves a field inversion Z^{-1} mod p, which has to be computed by the CPU, too. In theory, after obtaining $\frac{1}{Z}$, the CPU still needs to compute 4 modular multiplications over $\mathbb{F}_p$: $\frac{1}{Z^2} = \frac{1}{Z}\frac{1}{Z}$, $x = \frac{1}{Z^2}X$, $\frac{1}{Z^3} = \frac{1}{Z^2}\frac{1}{Z}$, and $y = \frac{1}{Z^3}Y$. However, since Algs. 1 & 2 only use x_1 but not y_1, we take a shortcut by having the GPU additionally compute Z^2 for the CPU; then, an implicit inversion needs the CPU to compute $\frac{1}{Z^2}$ and $x = \frac{1}{Z^2}X$ only, which correspond to 3+1 =4 modular multiplications.

Since the global memory, via which the CPU and the GPU exchange data (cf. Section II-C), has significant access latency, we need to minimize data exchange between the two processors. Therefore, in Alg. 1, after the PM result is obtained by the GPU and x_1 is obtained by the CPU at step 2, we have the CPU continue steps 3 and 4. A fringe benefit of doing so is that the GPU will not encounter the judgment on either r or s. In a signing operation the CPU performs 9 modular multiplications: 4 at step 2 (implicit inversion), and 3+1+1 at step 4 (for k^{-1}, dr, s, respectively). Likewise, in Alg. 2, after x_1 is obtained by the CPU at step 4, we have the CPU continue the rest of the work. A verifying operation also costs the CPU 9 modular multiplications: 3 at step 2 (explicit inversion), 2 at step 3, and 4 at step 4 (implicit inversion).

H. Nonce Generation and Utilization

Many signature schemes involve one-time random numbers known as nonces. Next, we look at step 1 in Alg. 1, where a once k should be selected with care [26], [27] so that it does not leads to the leakage of the signing key d (also recall the discussion on the side-channel attack in Section III-A).

There are several ways for generating an appropriate k. For example, it can be directly extracted from a physical noise source. However, such a hardware random number generator is relatively slow; it is inapplicable to high-throughput environments, for which Guess is intended. Another approach is deterministically deriving k from the private key d and the message digest e [28]. However, this trick conflicts with our vision that Guess is designed to be a *universal* signature server (cf. Section I-D), for which a reliable random number generator is supposed to be a general component. As a result, we adopt a trade-off (possibly the best practice): Guess obtains nonces from a deterministic random bit generator such as Hash_DRBG [29, Sec. 10.1.1], with its seed periodically sampled from a hardware random number generator. We implement NIST's Hash_DRBG on the more compute-capable GPU so that the nonces are generated massively. Therefore, in Alg. 1, the GPU supplies k in addition to kG to the CPU.

In Alg. 1, our kG (but not the one in [28]) is independent of the input d or e, and thus can be computed offline (i.e., beforehand) by the GPU (likewise, the CPU can prepare r and k^{-1}; such preparation does not increase Guess' throughput but decreases the service latency). Actually, we find that even when Guess handles signing requests with full throughput, the bottleneck lies in network I/O rather than the GPU. That is, no matter how intensive the signing requests are, the GPU can always afford the time to prepare k's and kG's, and they can be prepared more quickly than the CPU consumes them. This leads to an interesting fact that no kernel needs to be invoked *online* when Guess handles signing requests (we will be more specific in the next section). Besides favoring the performance, such offline handling significantly decouples Guess' signing processing from service requests, further deterring side-channel attacks.

This section revolves around implementing ECDSA, and our technique can be directly applied to other ECC schemes (e.g., the entire SM2 suite from China). To conclude this section, we briefly summarize our ECDSA implementation as follows. Only the most computation-heavy primitives, PMs, including first generating the nonce k in the fixed point multiplication kG, run on the GPU, while all the other primitives, including (explicit and implicit) modular inversions and certain modular multiplications, run on the CPU. From the CPU's perspective, only two kernels (cf. Section II-C) are (repeatedly) invoked, one for the fixed point multiplication (underlined in Alg. 1), another for the multiple point multiplication (underlined in Alg. 2). This complies with the GPU-accelerated computing model (cf. Section I-C) that bottlenecks of an application are moved to the GPU while the rest still runs on the CPU.

IV. System Architecture

In the previous section, we customize and optimize high/low-level algorithms to maximize Guess' computing power for the target signature scheme, ECDSA with Curve P-256. To build a high-performance signature server, we also

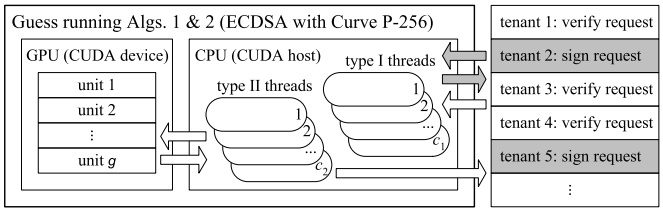

Fig. 2. The system architecture of Guess, which connects with and provides signature generation/verification services to multiple tenants. For signature generation (online flow indicated by hollow arrows in grey), type I threads employ k's and kG's prepared offline by type II threads (essentially by the GPU) to directly respond to requests. Signature verification requests (flow indicated by hollow arrows in white) are handled over from type I threads to type II threads that logically utilize the GPU as g units for online processing.

need to maximize Guess' capability to deliver its power to network tenants. Next, we present Guess' system architecture and the associated resource allocation and scheduling, which are independent of the target scheme and potentially applicable to implementing other forms of SIGaaS.

Fig. 2 illustrates the system architecture of Guess featuring a GPU as the cryptographic accelerator. There are two types of CPU threads: c_1 threads of type I, and c_2 threads of type II. Type I threads receive, classify, and queue[3] service requests from the tenants. For signing requests, type I threads also execute Alg. 1 and directly reply to the tenants, utilizing the k's and kG's prepared offline by type II threads (essentially by the GPU, cf. Section III-H). Verifying requests are then handed over to type II threads, which perform Alg. 2, invoke kernels online, and then send the responses to tenants.

Concerning signature verification, one problem arises for legacy GPUs that are only capable of executing kernels one by one: arriving requests are queued and then processed in batch by one kernel, and thus a service request arriving just after the kernel invocation has to wait until the GPU has completed one batch of processing and returned the service responses. We call the time a kernel needs for service computing the kernel latency. From the perspective of type II threads, the GPU continually goes offline and an "unlucky" request has to queue for up to the kernel latency until it may be moved into a processing batch. This needlessly increases the service latency for the requesting tenant. Should Guess support multiple (say, N_s) target schemes, the average *queueing* time would be $\frac{N_s}{2}$ times the kernel latency, which may be much more than the actual *processing* time.

Fortunately, a modern GPU can execute multiple kernels concurrently [11, Sec. 3.2.5.2]. We exploit this feature and allocate equal resources to g portions so that one physical GPU is logically partitioned into (and utilized as) g separate units, each serving as a virtual (but only $\frac{1}{g}$ powerful) GPU, as shown in Fig. 2. Such partition is applied to signature verification as well as generation; it does not improve Guess' throughput, but as type II threads can invoke kernels much more frequently, the average queueing time of service requests is significantly lowered (particularly so when N_s target schemes are supported). On our GPU, the hardware limit for g is 32 [11, Appendix G.1]; $g = 15$ might be a reasonable choice since our GPU has 15 SMs (cf. Section II-C).

A system architecture similar to ours is in [5]; our type I / II thread roughly corresponds to a "worker"/"GPU-interfacing" thread there. However, Guess is different from [5] in that instead of only one we launch c_2 type II threads. This is similar to the way we virtualize one GPU as g units: if the GPU works with only one type II thread of the CPU, from the GPU's perspective it is continually "left idle". Note that besides calling the GPU the CPU is also burdened with tasks like 9 modular multiplications per ECDSA operation (cf. Section III-G) and data transfers (including queue access and data copy between the CPU and the GPU). Keeping the GPU as busy as possible (instead of leaving it idle) is a key to good performance [10, Sec. 10]. To this end, we launch c_2 independent type II threads so that for the GPU the physical CPU appears as c_2 units, each serving as a virtual CPU. Likewise, we launch c_1 type I threads to work with the c_2 type II threads.

Launching multiple type II CPU threads also enables us to overlap data transfer with kernel execution [11, Sec. 3.2.5.3] to "hide" data transfers [10, Sec. 9.1.2]. The CPU units can either access the same GPU unit in turn, or simultaneously access different GPU units. We call it c_2-to-g CPU-GPU coordination. There may be several ways to implement the c_2-to-g coordination. For simplicity and stability, for each type II thread we allocate a dedicated buffer in the device's global memory (so that no global memory is shared between any two type II threads), and we have a type II thread invoke only one kernel at a time, whose global memory access (e.g., for the PCT involved in Table V) is confined in the invoking thread's buffer. Thus, c_2 buffers are needed, each of which may or may not be accessed by one of the g GPU units. To make sure all g GPU units can be simultaneously activated (e.g., for maximum throughput), we set $c_2 \geq g$.

V. Performance Evaluation

We evaluate the performance of our GPU-accelerated universal elliptic-curve signature server with an emulated EPC environment, where an H3C S5800-56C Ethernet switch (with four 10Gb ports) connects 3 tenants to Guess. A tenant is a heavily loaded application server that outsources to Guess intensive signature generation and/or verification assignments. We emulate each tenant with a computer of the same hardware configuration with Guess (e.g., with a 10Gb Ethernet controller). When combined together, the 3 tenants should impose sufficient burden on Guess for stress testing. To this end, each tenant launches t independent threads. First, each thread makes a TCP connection (which it does not close) to Guess. Then, each thread sends a request packet to Guess and waits until it receives a service reply, and repeats this send-receive paradigm without a stop. This may emulate the scenario that a tenant (e.g., a payment platform) simultaneously serves t extremely active clients (e.g., buyers). From Guess' perspective, there are always $C(= 3t)$ connections from the tenants, which are invisible to its GPU but accessed by type I threads for reading and type I / II threads for writing (cf. Fig. 2).

[3]As to be shown shortly, signing requests are processed by type I threads almost instantly, and thus queueing is only applied to verifying requests.

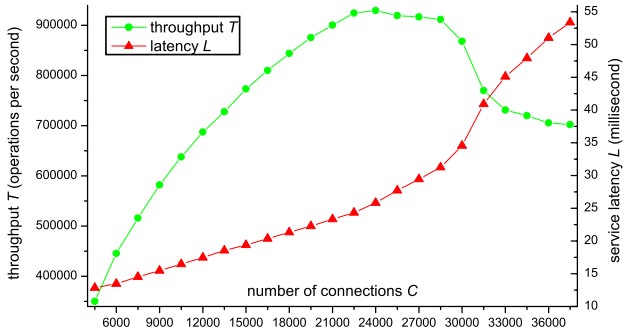

Fig. 3. Verification throughput and latency for the parameter combination ($g=15$, $c_2=25$, $c_1=12$, $N_b=4$, $N_t=256$), Guess' default configuration.

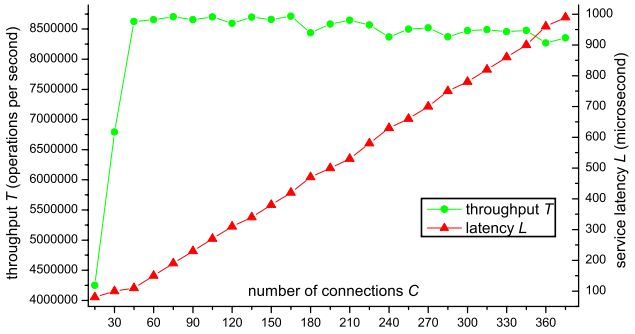

Fig. 4. Generation throughput and latency (explicit key specification) with 22 signing requests per packet, using the same configuration for verification.

The two metrics for performance evaluation are throughput and latency (cf. Section I-D). Guess periodically monitors its throughput T, which is the number of transactions completed in one period (e.g., 3s) divided by the period. On the other hand, a tenant locally averages the service latency L, which is the waiting interval from sending a request packet to receiving a reply. We investigate how T and L change when we increase C, the number of active TCP connections to Guess.

A. Signature Verification

Since signature verification is intrinsically more expensive than generation, we begin with the performance evaluation on the former. In our tests, the signature in each verifying request sent to Guess is generated by a tenant with OpenSSL employing a distinct private key; Guess follows the standard individual verification (i.e., Alg. 2) and no batch verification [30], [31] is applied. We need first to identify the parameter combination for Guess to reach its maximum verifying throughput. Then, for ease of use (i.e., zero user configuration), all parameters will be fixed no matter whether the users employ Guess for verifying or signing, unless there is a platform upgrade.

The key parameters concerning our system architecture (cf. Fig. 2) are g and c_2; we test each combination ($g \leq 32$, $c_2 \geq g$) (cf. Section IV) and find that ($g = 15, c_2 = 25$) seems to be optimal for verifying. A less sensitive parameter is c_1; we also fine tune it and 12 is chosen. Regarding CUDA programming, the most important parameters are the number of blocks N_b and the number of threads per block N_t (cf. Section II-C). We have each type II thread designate ($N_b = 4$, $N_t = 256$) when invoking a kernel.

For the above configuration, we test T and L against C, and the experimental results are depicted in the double Y-axis Fig. 3. We observe that the service latency increases with C, but the throughput reaches it maximum $T_v = 9.29 \times 10^5$ OPS only at $C = 24000$ (when there are 8000 connections from each tenant); at this point the measured latency is $L_v = 25.82$ ms. Guess allows users to trade throughput for latency. For example, when $C = 16500$, the latency reduces to only 20.37 ms but the throughput is still as high as 8.10×10^5 OPS.

B. Signature Generation

For signature generation, we could simply preload Guess with the private key of every tenant in the EPC, and Guess can easily identify which key to use according to the source of a signing request. This is the preferred *implicit* key specification. For stress testing, however, we push the envelop by associating each signing request with a different private key (from Guess' perspective it is serving numerous distinct tenants). Specifically, each of our "physical" tenants sends in clear to Guess a random 256-bit private key d in addition to a random 256-bit message digest e (cf. Alg. 1). Clearly, such *explicit* key specification doubles the payload, costs Guess more memory and bandwidth, and adversely affects the performance; it is a for-test-only overkill and should never be adopted for real deployment. In other words, what we obtain from the stress testing below is just a worst-case signing performance.

As mentioned earlier, we follow the same configuration optimized for verifying. Even so, owing to the offline handling (cf. Section III-H) Guess turns out to be extremely fast for signing: the latency L is typical 0.2 ms, implying that type I threads reply to requests almost instantly (cf. Fig. 2), and that the throughput T can be observable even if there is only $C = 1$ connection from the tenants. An attending problem is that we find it a bit difficult for T to enlarge. For example, when $C = 315$, T reaches its maximum value 1.46×10^6 OPS, which is merely on par with the previous T_v.

We then realize that we can increase T by having one packet carry more than one signing request (d, e), which is of $\frac{256+256}{8} = 64$ bytes. For Ethernet, the maximum transmission unit size is 1500 bytes, and the minimum sizes of IP and TCP headers are both 20 bytes; each packet can accommodate $\lfloor \frac{1460}{64} \rfloor = 22$ requests from one tenant (besides an indicator of the type of request, herein "signing"). Such multiplexing not only reduces the "header costs" for the tenant, but also proves to be effective in unleashing Guess' computing power. For 22 requests per packet, we test T and L against C and depict the results in the double Y-axis Fig. 4. We find that L still increases with C, but T surges and quickly becomes stable. When $C = 165$, T reaches it maximum $T_s = 8.71 \times 10^6$ OPS; at this point the measured latency is $L_s = 0.42$ ms (420 μs). Users can still trade throughput for latency. For example, when $C = 45$ the latency reduces to merely 0.11 ms while the throughput is still as high as 8.62×10^6 OPS.

Again note that the above T_s and L_s only indicate a worst-case performance. As implicit key specification is applicable in real deployment, the tenants can embed more requests per

TABLE VI
THROUGHPUTS OF THE UNKNOWN POINT MULTIPLICATION REPORTED IN THE LITERATURE (SORTED IN CHRONOLOGICAL ORDER)

Throughput (OPS)	Key length (bits)	Device of accelerator
24,700 [33]	256 (NIST curve)	FPGA (Virtex-4)
1,413 [34]	224 (NIST curve)	GPU (GeForce 8800 GTS)
1,620 [35]	256	GPU (GeForce 9800 GX2)
3,138 [12]	224 (NIST curve)	GPU (GeForce 8800 GTS)
9,827 [36]	224	GPU (GeForce GTX 285)
79,198 [3]	224 (NIST curve)	GPU (GeForce GTX 295)
290,535 [3]	224 (NIST curve)	GPU (GeForce GTX 580)
2,600 [37]	256	FPGA (Virtex-5)
47,000 [38]	224	GPU (GeForce GTX 680)
115,200 [4]	224	GPU (GeForce GTX 285)

packet for higher throughput, and Guess can pre-compute the modular multiplication dr in step 4 of Alg. 1 for lower latency.

C. Comparison With Related Work

Rather than providing a cloud service deliverable to network tenants, many commercial products and research prototypes only implement ECC in a simple "resident" manner (thus comparing their local throughput with Guess' service delivery is not fair for Guess). For example, Freescale[4] manufactured a family of high-throughput cryptographic coprocessors [32] that can be integrated to a host via PCIe connection, among which C293 is the fastest for ECC-256, achieving 68,321 OPS for signing and 49,935 OPS for verifying. By contrast, Guess is 127.5/18.6 times faster for signing/verifying, even if we do not consider the loss of computing power due to network delivery. For another example, performances of recent research efforts are summarized in Table VI, but all of them implement merely the unknown point multiplication (cf. Section III-E) rather than the complete ECDSA verification. Among those, the highest throughput is 2.9×10^5 ECC-224 PMs per second locally measured by [3], while our Guess can verify $T_v = 9.29 \times 10^5$ ECC-256 signatures per second outsourced by tenants.

Luna SA 7000 [22] and NShield Connect 6000+ [8] are the few signature servers on the market (hence a relatively fair comparison is applicable) and they are optimized for ECC-256, with signing throughputs of 1000 and 2400 OPS, respectively (verifying throughput for either product is unavailable). Guess is at least several thousand times faster than either of them as even in the worst case $T_s = 8.71 \times 10^6$ OPS.[5]

Finally, let us get back to the record (cf. Section I) recently set by Alipay, which in 1 second may need to handle up to 85,900 signature generations and 171,800 verifications. Even if all the influx is offloaded to Guess, the worst-case processing time is about $\frac{85,900}{T_s} + \frac{171,800}{T_v} = 0.19 < 1$ s, implying Guess is capable of serving busy tenants in real time, even if they are the most saturated application servers in the world.

VI. CONCLUDING REMARKS

In this paper, we have demonstrated our exhaustive experience with a concrete case study on SIGaaS. Specifically, we have developed a tangible GPU-accelerated universal elliptic-curve signature server called Guess for short.

Our primary contribution is a novel, systematic, and inclusive implementation of ECDSA, turning cryptographic theory into productivity on off-the-shelf processors. For example, we have proposed high-level algorithms for the PM primitives, which are the bottleneck of ECDSA and other ECC schemes. For another example, space-efficient low-level algorithms have been devised to bolster our one-thread-per-task approach, so that we can launch more GPU threads for higher throughput. Besides customizing and optimizing various algorithms for our platform to maximize its computing power, we have also leveraged efficient resource management to provide Guess' computing power to network tenants with minimum loss. Field experiments have shown that Guess achieves significantly higher performance than existent prototypes and products.

Guess is implemented with software on a general platform that is fairly affordable and easily upgradable, implying that further performance improvement and functionality extension are feasible. This facilitates scalable and flexible cloud service provision. In fact, Guess readily supports various categories of ECC schemes like digital signature, key agreement, and encryption. We hope Guess can serve as a proof of concept that securing Internet transactions with strong digital signatures (e.g., ECDSA with 128-bit security) can be far more cost-effective in practice than one used to reckon. We also hope our endeavor can shed light on future research and inspire more case studies on SIGaaS as well as other forms of SECaaS.

Our future work includes rigorous tests on Guess' resilience to side-channel attacks. It would also be interesting to implement signatures on Guess with respect to other elliptic curves.

[4]One of the world's first semiconductor companies, acquired in late 2015 by NXP Semiconductors, one of the world's top 20 semiconductor sales leaders.

[5]Nevertheless, Guess is more energy-consuming. The typical power consumption of Luna SA 7000 is 155W, but that of Guess is as large as 420W.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] (Nov. 2015). *The Numbers Behind Alibaba's Singles Day*. [Online]. Available: http://finance.yahoo.com/news/numbers-behind-alibaba-singles-day-202157

[2] V. Varadharajan and U. Tupakula, "Security as a service model for cloud environment," *IEEE Trans. Netw. Service Manage.*, vol. 11, no. 1, pp. 60–75, Mar. 2014.

[3] J. W. Bos, "Low-latency elliptic curve scalar multiplication," *Int. J. Parallel Program.*, vol. 40, no. 5, pp. 532–550, Oct. 2012.

[4] S. Cui, J. Großschädl, Z. Liu, and Q. Xu, "High-speed elliptic curve cryptography on the NVIDIA GT200 graphics processing unit," in *Proc. ISPEC*, May 2014, pp. 202–216.

[5] K. Jang, S. Han, S. Han, S. B. Moon, and K. Park, "SSLShader: Cheap SSL acceleration with commodity processors," in *Proc. USENIX NSDI*, Apr. 2011, pp. 1–14.

[6] NIST. (Jul. 2013). *Digital Signature Standard (DSS) FIPS 186-4*. [Online]. Available: http://csrc.nist.gov/publications/PubsFIPS.html

[7] NIST. (Jul. 2012). *Recommendation for Key Management—Part 1: General (Revision 3), SP 800-57 Part 1—Rev. 3*. [Online]. Available: http://csrc.nist.gov/publications/PubsSPs.html

[8] Thales e-Security. *nShield Connect Data Sheet*, accessed Feb. 2006. [Online]. Available: https://www.thales-esecurity.com/products-and-services/products-and-services/hardware-security-modules/general-purpose-hsms/nshield-connect/

[9] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[10] NVIDIA. *CUDA C Best Practices Guide*, accessed Feb. 2006. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/
[11] NVIDIA. *CUDA C Programming Guide*, accessed Feb. 2006. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/
[12] S. Antão, J.-C. Bajard, and L. Sousa, "Elliptic curve point multiplication on GPUs," in *Proc. 21st IEEE Int. Conf. Appl., Specific Syst. Archit. Process. (ASAP)*, Jul. 2010, pp. 192–199.
[13] F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Exploiting the floating-point computing power of GPUs for RSA," in *Proc. ISC*, Oct. 2014, pp. 198–215.
[14] F. Zheng, W. Pan, J. Lin, J. Jing, and Y. Zhao, "Exploiting the potential of GPUs for modular multiplication in ECC," in *Proc. WISA*, Jan. 2015, pp. 295–306.
[15] M. Rivain. (2011). *Fast and Regular Algorithms for Scalar Multiplication Over Elliptic Curves, IACR Cryptology ePrint Archive 2011/338*. [Online]. Available: http://eprint.iacr.org/2011/338.pdf
[16] J. W. Kim, J. Seo, J. Hong, K. Park, and S.-R. Kim, "High-speed parallel implementations of the rainbow method in a heterogeneous system," in *Proc. INDOCRYPT*, Dec. 2012, pp. 303–316.
[17] P. Karpman, T. Peyrin, and M. Stevens, "Practical free-start collision attacks on 76-step SHA-1," in *Proc. CRYPTO*, Aug. 2015, pp. 623–642.
[18] EVGA. *EVGA GeForce GTX 780 Ti Superclocked*, accessed Feb. 2006. [Online]. Available: http://www.evga.com/articles/00795/#2883
[19] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. New York, NY, USA: Springer, 2004.
[20] (Dec. 2010). *Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves—Part 2: Digital Signature Algorithm*. [Online]. Available: http://www.oscca.gov.cn/UpFile/2010122214822692.pdf
[21] Intel. *Intel Xeon Processor E5-2697 v2*, accessed Feb. 2006. [Online]. Available: http://ark.intel.com/products/75283/Intel-Xeon-Processor-E5-2697-v2
[22] SafeNet and Gemalto. *Product Brief: Gemalto Safenet Luna SA Hardware Security Module*, accessed Feb. 2006. [Online]. Available: http://www.safenet-inc.com/WorkArea/linkit.aspx?LinkIdentifier=id&Item
[23] T. Hansen, *US Secure Hash Algorithms (SHA and SHA-Based HMAC and HKDF)*, document RFC 6234, May 2011.
[24] Y. Yarom and N. Benger. (2014). *Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-Channel Attack, IACR Cryptology ePrint Archive 2014/140*. [Online]. Available: http://eprint.iacr.org/2014/140.pdf
[25] J. van de Pol, N. P. Smart, and Y. Yarom, "Just a little bit more," in *Proc. CT-RSA*, Mar. 2015, pp. 3–21.
[26] J.-C. Faugère, C. Goyet, and G. Renault, "Attacking (EC)DSA given only an implicit hint," in *Proc. SAC*, Aug. 2012, pp. 252–274.
[27] J. Chen, M. Liu, H. Li, and H. Shi, "Mind your nonces moving: Template-based partially-sharing nonces attack on SM2 digital signature algorithm," in *Proc. ASIACCS*, Apr. 2015, pp. 609–614.
[28] T. Pornin, *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*, document RFC 6979, Aug. 2013.
[29] NIST. (Jun. 2015). *Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revision 1) SP 800-90A—Rev.1*. [Online]. Available: http://csrc.nist.gov/publications/PubsSPs.html
[30] S. Karati and A. Das, "Faster batch verification of standard ECDSA signatures using summation polynomials," in *Proc. ACNS*, June 2014, pp. 438–456.
[31] S. Karati, A. Das, and D. Roychoudhury, "Randomized batch verification of standard ECDSA signatures," in *Proc. SPACE*, Oct. 2014, pp. 237–255.
[32] NXP Semiconductors. *Freescale C29x Crypto Coprocessor Family Product Brief—Rev.1, 07/2015*, accessed Feb. 2006. [Online]. Available: http://cache.nxp.com/files/32bit/doc/prod_brief/C29xPB.pdf
[33] T. Güneysu and C. Paar, "Ultra high performance ECC over NIST primes on commercial FPGAs," in *Proc. CHES*, Aug. 2008, pp. 62–78.
[34] R. Szerwinski and T. Güneysu, "Exploiting the power of GPUs for asymmetric cryptography," in *Proc. CHES*, Aug. 2008, pp. 79–99.
[35] P. Giorgi, T. Izard, and A. Tisserand, "Comparison of modular arithmetic algorithms on GPUs," in *Proc. ParCo*, Sep. 2009, pp. 315–322.
[36] S. Antão, J.-C. Bajard, and L. Sousa, "RNS-based elliptic curve point multiplication for massive parallel architectures," *Comput. J.*, vol. 55, no. 5, pp. 629–647, May 2012.
[37] Y. Ma, Z. Liu, W. Pan, and J. Jing, "A high-speed elliptic curve cryptographic processor for generic curves over GF(p)," in *Proc. SAC*, Aug. 2013, pp. 421–437.
[38] S. Pu and J.-C. Liu, "EAGL: An elliptic curve arithmetic GPU-based library for bilinear pairing," in *Proc. Pairing*, Nov. 2013, pp. 1–19.

Wuqiong Pan received the B.E. degree from Tsinghua University and the Ph.D. degree from the University of Chinese Academy of Sciences in 2008 and 2014, respectively. He is currently an Assistant Professor with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences. His research interests include hardware security module, virtual machine security, and software guard extensions.

Fangyu Zheng received the B.E. degree from the University of Science and Technology of China, and the Ph.D. degree from the University of Chinese Academy of Sciences in 2011 and 2016, respectively. He is currently an Assistant Professor with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences. His research interests include applied cryptography and high performance computing.

Yuan Zhao received the B.S. and M.E. degrees from Peking University in 2009 and 2013, respectively. He is currently pursuing the Ph.D. degree in computer science with the Institute of Information Engineering, Chinese Academy of Sciences. His research interests include cryptography engineering and side-channel attacks.

Wen-Tao Zhu (SM'16) received the B.E. and Ph.D. degrees from the Department of Electronic Engineering and Information Science, University of Science and Technology of China in 1999 and 2004, respectively. Since July 2004, he has been a faculty member with State Key Laboratory of Information Security, which is now part of Institute of Information Engineering, Chinese Academy of Sciences (IIECAS), where he has been a professor since October 2011. His research interests include network and information security and applied cryptography. Since 2011, he has been serving on the Editorial Board of the *Journal of Network and Computer Applications* (Elsevier).

Jiwu Jing received the B.E. degree from the Department of Electronics Engineering, Tsinghua University, in 1987, and the M.Sc. and Ph.D. degrees from Graduate School, University of Science and Technology of China, in 1990 and 2003, respectively. He is currently a Professor and a Deputy Director of the Institute of Information Engineering with the Chinese Academy of Sciences (CAS). He is also the Director of Data Assurance and Communication Security, CAS. His research interests include public key infrastructure, identity management, fault tolerance, mobile security, information system, and network protection. His research has won the National Science and Technology Progress Awards, Science and Technology Progress Award of the Chinese Ministry of Electronics Industry, and the Science and Technology Progress Award of CAS.