GPS: Integration of Graphene, PALISADE, and SGX for Large-scale Aggregations of Distributed Data

Jonathan Takeshita
jtakeshi@nd.edu
University of Notre Dame
Notre Dame, Indiana, USA

Colin McKechney
cmckechn@nd.edu
University of Notre Dame
Notre Dame, Indiana, USA

Justin Pajak
jpajak@nd.edu
University of Notre Dame
Notre Dame, Indiana, USA

Antonis Papadimitriou
apapadimitriou@dualitytech.com
Duality Technologies
Newark, New Jersey, USA

Ryan Karl
rkarl@nd.edu
University of Notre Dame
Notre Dame, Indiana, USA

Taeho Jung
tjung@nd.edu
University of Notre Dame
Notre Dame, Indiana, USA

ABSTRACT

Secure computing methods such as fully homomorphic encryption and hardware solutions such as Intel Software Guard Extension (SGX) have been applied to provide security for user input in privacy-oriented computation outsourcing. Fully homomorphic encryption is amenable to parallelization and hardware acceleration to improve its scalability and latency, but is limited in the complexity of functions it can efficiently evaluate. SGX is capable of arbitrarily complex calculations, but due to expensive memory paging and context switches, computations in SGX are bound by practical limits. These limitations make either of fully homomorphic encryption or SGX alone unsuitable for large-scale multi-user computations with complex intermediate calculations.

In this paper, we present GPS, a novel framework integrating the Graphene, PALISADE, and SGX technologies. GPS combines the scalability of homomorphic encryption with the arbitrary computational abilities of SGX, forming a more functional and efficient system for outsourced secure computations with large numbers of users. We implement GPS using linear regression training as an instantiation, and our experimental results indicate a base speedup of 1.03x to 8.69x (depending on computation parameters) over an SGX-only linear regression training without multithreading or hardware acceleration. Experiments and projections show improvements over the SGX-only training of 3.28x to 10.43x using multithreading and 4.99x to 12.67 with GPU acceleration.

1 INTRODUCTION

1.1 Scale and Complexity of Modern Data Analytics

Modern computing scenarios have an urgent need for secure computation over private user data. Fields including healthcare, education, finance, genomics, and advertising all have some need to protect the confidentiality of users' private inputs and attributes, while being able to make use of that data. Traditional approaches of gathering all users' unencrypted data at datacenters are vulnerable to data breaches, as shown by frequent accidents in the last decade [19]. An alternative is to collect encrypted data from users to avoid such risks. In other words, collecting encrypted data from users for secure computations at the aggregator's server is a particularly interesting case for the industry.

Besides ensuring users' security and privacy, the modern era presents new challenges in processing user data. The first is the scale of the computation. Google performs 5.6 billion searches per day [66] and serves nearly 5 billion videos on YouTube daily [1], while Facebook claims approximately 1.9 billion daily active users [4]; the behavior of the users such as content viewed or liked, ads clicked on, or duration of a page visit are all data that can be analyzed in aggregate. The second is in the complexity of the computations that are undertaken at this scale. Analysis on user data is not simply numerical calculation, but runs a gamut of complexity from simple if-else statements to activation functions and matrix inverses. This complexity is inherent in the applications such as distributed machine learning, statistical calculations, and voting and consensus protocols, that are notable use cases for privacy-preserving computing. When taken together, this set of issues is extremely challenging, even with the wide variety of privacy-preserving computational tools used in both research and industry.

Several broad categories of secure computation can be applied for multi-user computations, including Homomorphic Encryption (HE), Trusted Execution Environments (TEEs), and Secure Multi-party Computation (MPC). HE allows computation to take place over encrypted data, separating knowledge and computation and thus hiding a users' data from the eyes of a cloud computing service, however, its overhead in both computation and communication can be prohibitively high, and HE faces some practical issues with nonarithmetic computation. Further, HE needs either a trusted decryptor or the use of highly complex and expensive multiparty

HE schemes [8]. MPC allows different users to jointly compute a function over everyone's input without anyone learning others' inputs, however it does not scale well with the number of users due to the large communication overhead. TEEs are a hardware solution to provide a secure execution environment safe from spying or tampering against even a malicious operating system or hypervisor, but they have difficulties when the scale of data aggregation becomes large due to high paging overhead and limited memory space, especially at the scale demanded in the industry by large companies such as Facebook [5]. All of these approaches are able to protect data owners' information, satisfying laws such as GDPR, HIPAA, and FERPA. Any of these technologies can be applied for use in computation over aggregated user data, however, each one has its own disadvantages for the extreme scale and high complexity presented in modern data analytics. Because of the difficulty in applying any of these approaches to massively large and complex aggregate computation, we instead consider the construction of a hybrid approach. Traditionally, the research areas of SGX and HE have been considered as two disjoint areas, because one is hardware-based the other is theory-based. This paper is motivated by the complementary strengths and limitations of SGX and HE.

1.2 Limitations of Previous Approaches in Secure Computing at High Complexity and Scale

SGX provides fast and trusted arbitrary computation for smaller workloads, outstripping HE for many computations; however, SGX faces serious difficulties at scale. In particular, the practical memory space of SGX is limited to about 96MB of physical memory and 64GB with paging [3], with significant overhead incurred by paging due to the need to encrypt/decrypt pages and context switches. In the case of large-scale aggregations of distributed data, an SGX application needs to read many inputs from a large number of users, which will incur a large overhead due to paging. These disadvantages make SGX difficult to apply for large-scale computations involving many users' aggregated inputs. While full/total memory encryption technologies are planned for rollout to consumer CPUs, they are not currently mature and widely available [46], thus the performance impact and other downsides of total memory encryption technologies are not yet well-known.

HE cryptosystems allow computation over encrypted data. The most functional HE schemes are Fully Homomorphic Encryption (FHE) schemes, which allow arbitrary computation [15, 22, 37]. Although HE incurs extra overhead due to ciphertext expansions, it is not limited in scale as SGX is. In practice, most FHE schemes are implemented as their Somewhat Homomorphic Encryption (SHE) versions, which allow leveled computations bounded by multiplicative depth. Highly-optimized SHE can be very efficient and show high throughput [44], with a high potential for parallelization and optimizations [11, 42, 54]. However, it has other limitations. SHE's overhead grows with the multiplicative depth supported, and this constraint limits the class of computations that can be done efficiently. Also, functions involving branching (e.g., if-else, thresholds) on encrypted data cannot be computed efficiently, as they are non-arithmetic.
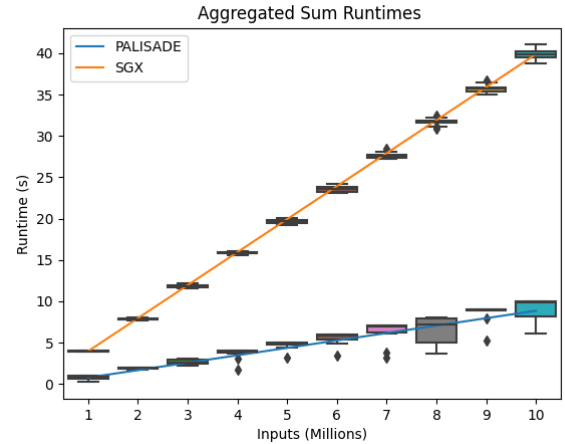
Figure 1: Comparing Simple Sum Calculations. (PALISADE implementation used depth-1 CKKS w/ $N = 512$, no packing.)

In summary, SGX supports smaller general secure computations efficiently but is limited in its performance at scale. HE enjoys better parallelism compared to SGX and lacks scale-limiting factors such as SGX's paging overhead, but supports only simple arithmetic functions with limited multiplicative depths efficiently. Computations in SGX with good memory locality and a low memory footprint can be run much more quickly than the same computation run with HE [80, 87]. However, this may not be the case in some scenarios, such as when processing separate inputs from a large number of users. This is shown in our preliminary experiments where we compared a simple additive aggregation (sum) with inputs from a large number of users (Figure 1). In the case of PALISADE, users upload ciphertexts of SHE upon which homomorphic additions are performed. In the case of SGX, they upload AES-encrypted ciphertexts with individual session keys shared with an SGX enclave, which are decrypted inside the SGX enclave to perform the sum calculation. The trend of runtime as the input size increased with homomorphic encryption using the PALISADE was much better than SGX. These experiments were run in the same environment described in Section 5.1, and measured the calculation of a sum of millions of user inputs.[1] Note that HE outperforms SGX even without using batching, which can greatly improve throughput. This is due to the overhead of paging many users' inputs into SGX with context switches. This result from our preliminary experiment shows the case where HE is superior to SGX under certain circumstances, which indicates the need for using integrated solutions for scalable secure computation solutions. For example, HE can be used to overcome the limitations of SGX in dealing with large-scale inputs from users, and the SGX can still be leveraged to provide functionality that HE schemes have difficulties with.

There exist some lines of research seeking to combine the capabilities of cloud-based HE and TEEs to cover each strategy's disadvantages [20, 25, 34, 38, 70, 77, 87], and previous work in this area

---

[1]The source code of these experiments can be found at https://anonymous.4open.science/r/Sum_SGX-52DD/ and https://anonymous.4open.science/r/homomorphic-sum-922F/.
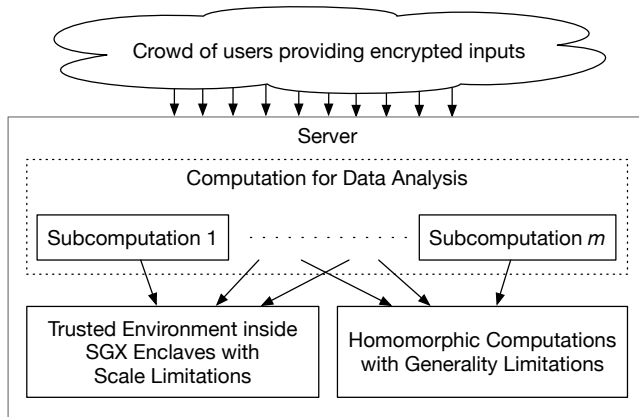
Figure 2: Overview of Our Integration

has successfully implemented small-scale proof-of-concept combinations of HE and TEEs. In order to further explore the possibilities of combining HE and TEEs for high-scale workloads, there is a need to implement and integrate state-of-the-art cryptographic schemes and libraries with TEEs. Prior related work combining TEEs and cloud-based HE [20, 70] relied upon customizing or freshly implementing HE functionality to allow porting to Intel SGX. However, this makes easily using the full range of functionality of existing state-of-the-art HE libraries difficult, as this process has some difficulties; restrictions on external dependencies and in-library use of I/O and system calls complicates such ports. Further, application programming in the SGX model is tedious, making this strategy undesirable for future lines of research. Also, prior work in combining cryptography and SGX has not focused on the scenario of many users on large scales, which is a case that is especially salient in today's world of outsourced computations.

1.3 Our Approach

In this paper, we present **GPS**, a novel integration of homomorphic encryption schemes and TEEs into a pipeline of secure computations that benefits from one's strengths that mitigate the other's limitations and vice versa, for securely computing a function where inputs are coming from a large number of users. A library OS for unmodified applications, **G**raphene [80], is used to integrate a HE library, **P**ALISADE [65], and a popular implementation of TEEs, Intel **S**GX [27]. It divides a computation into several subcomputations (Figure 2), which can be performed by either the trusted computation inside SGX or by homomorphic evaluations of HE. GPS is formulated for an honest-but-curious adversary model, which is the most relevant choice for discussion of secure outsourced computation protocols. In such scenarios (e.g., outsourced HE computations), computation integrity provided by the SGX is not a goal of the protocol; rather, efficient and secure computation is considered. Similarly to other work combining SGX and cryptographic methods [25, 70], we do not consider side-channel attacks against the SGX or other practical vulnerabilities; these are out of scope and discussed at length in other work [12, 31, 40, 59, 60, 88].

Instead of following previous approaches in reimplementing HE for SGX, we thus instead explore the use of new containerization

tools for running SGX applications. In particular, we pioneer the use of the Graphene containerization framework [80] for integrating HE and SGX. Similarly to other containerization solutions such as Docker, Graphene provides a lightweight library OS to applications to allow unmodified binaries to run on various hardware. By applying Graphene, we can use the feature-rich PALISADE HE library, and write ordinary applications without having to manually program according to the SGX model. This work is the first to explore using containerization to allow SGX-assisted HE by integrating SGX and an existing feature-rich HE library, bypassing a rewrite of HE libraries and applications for SGX.

Properly integrating SGX and HE for optimal efficiency is challenging even with containerization. To do so, the strengths and disadvantages of different secure computation paradigms should be considered carefully in different applications to determine how a hybrid secure computation pipeline (Figure 2) can be constructed for the best efficiency. We first review the advantages and disadvantages of HE and TEEs in detail, and discuss their combination in previous research. This leads to a discussion of what work is most important to advance this line of research, and in particular our recommendations for what schemes and libraries are the best candidates for integration into SGX for future work. We then present some strategies for such integration, and also discuss what applications are well-suited for such a hybrid secure computation paradigm. Finally, we apply the methodologies in the linear regression training as a concrete instantiation of GPS, and present experiment results with the implementations.

1.4 Contributions

(1) We propose a novel secure computation paradigm that combines TEEs and HE into a pipeline of secure computation in large-scale applications, which can achieve the high efficiency that cannot be achieved by TEEs or HE alone.
(2) We analyze how a system using TEEs to provide assisted computing to HE can be best designed to take full advantage of the relative strengths and capabilities of both tools, and how computations can most advantageously be split between a TEE and HE.
(3) We present a concrete implementation with source code of our system using Graphene, PALISADE, and SGX that is easily reconfigurable for different schemes and applications. This is the first such system using containerization for convenient and efficient integration of SGX and existing HE libraries. We discuss our various choices in our system's design, justifying our choices of schemes and tools, and show our experimental results.

2 BACKGROUND

2.1 Homomorphic Encryption

Homomorphic Encryption (HE) schemes allow for some computation to take place on encrypted data. Fully Homomorphic Encryption (FHE) schemes allow arbitrary computation, but are practically limited by their complexity and the need to periodically refresh encrypted data between some operations. In practice, most FHE

schemes are implemented as their Somewhat Homomorphic Encryption (SHE) versions, which allow arbitrary computation up to some multiplicative depth.

The DGHV [82], TFHE [23], and FHEW [35] FHE schemes are FHE schemes that operate on single-bit plaintexts or batches thereof. DGHV is a very simple and intuitive scheme, and TFHE and FHEW are capable of highly efficient programmable bootstrapping to allow computation of arbitrary depth. However, having single-bit or bit-encoded plaintexts is highly cumbersome for many real-world applications. Implementations of these schemes are available [24, 26, 65], though some are not currently maintained.

The B/FV [37] and BGV [16] schemes are both FHE schemes that encrypt and perform homomorphic computation on elements of $\mathbb{Z}_t$, i.e., integral HE schemes. The similar CKKS scheme [22] uses a fixed-point encoding to do approximate encrypted arithmetic. Batching via the Chinese Remainder Theorem for polynomial rings allows packing thousands of operands into a single plaintext, greatly improving practical throughput for B/FV, BGV, and CKKS. Batched ciphertexts can use ciphertext rotation to permute the order of the packed elements. Many different libraries implement B/FV, BGV, and CKKS, including Microsoft SEAL [72], PALISADE [65], HElib [43], and Lattigo [58]. All of B/FV, BGV, and CKKS operate on polynomial rings. Ciphertexts are in $R_q = \mathbb{Z}_q/(x^N + 1)$ and plaintexts are in $R_t = \mathbb{Z}_t/(x^N + 1)$ for a ciphertext modulus degree N which is a power of two. Arithmetic on these rings can be accelerated by use of Residue Number System arithmetic [11, 42] and the Number-Theoretic Transform [54].

Using FHE has some pitfalls:

(1) Ciphertext expansion: In FHE schemes, the ciphertext may be much larger than the plaintext. As an example, VISE [25] showed ciphertext expansion factors of $2000\times$ for TFHE and up to $\approx 10^{12}\times$ for DGHV. (The expansion factors are more tolerable for B/FV, BGV, and CKKS with batching.) In scenarios where bandwidth between data owners and evaluators is limited, the size of FHE operands may pose a limitation. Further, ciphertext sizes increase commensurate to the complexity of the computations that a particular parameter setting supports, so higher utility incurs greater costs.

(2) Computational intensity: Operations on the large operands of FHE ciphertexts are computationally heavy. Homomorphic multiplication may take up to hundreds of milliseconds for larger parameters offering a greater multiplicative depth [87]. Smaller parameters can improve runtime at the cost of depth (discussed next), and batching can improve throughput, but the intensity is inherent in the mathematics.

(3) Depth: FHE schemes can only perform evaluation up to a certain multiplicative depth, at which point the expensive operation of bootstrapping is needed to refresh the ciphertext. Using SHE variants in practice avoids this overhead, but limits the possible evaluations in their multiplicative depth.

(4) Unfriendly applications: Some functionality such as branching or looping on encrypted data is not easily supported in FHE, restricting practical computations to those expressed in a purely arithmetic or logical fashion.

Careful parameter selection and optimization of the computation can mitigate ciphertext expansion and computation intensity.

Parallel computing, cloud computing, and hardware acceleration can also be used to accelerate FHE. However, for computations with high multiplicative depth or otherwise unfriendly computations, SGX may be a more effective approach.

2.2 TEEs and Intel SGX

Trusted Execution Environments (TEEs) are systems that enable trusted and secure computing even in the presence of a malicious host operating system. Code running in a TEE is free from tampering from other processes, whether at user or kernel level. The memory of the process is held in a secure enclave that is encrypted when paged out and decrypted when paged in, so even ring 0 processes cannot read it. TEEs can also encrypt/decrypt data using privately held keys, making their external communications undecipherable to the host OS. A TEE can also perform remote attestation to assure other parties that it is honestly running a given program. A widely used realization of a TEE is Intel SGX [27].

SGX also has pitfalls:

(1) Expensive paging: Due to the encryption/decryption on paging, paging data in/out of the protected memory enclave incurs a latency penalty [39].

(2) Memory limit: Intel SGX has a physical memory limit of 128MB, due to the limited size of Processor Reserved Memory [39]. The practical limit is closer to 90MB [27]; using a greater amount of memory may incur untenable overhead from frequent paging. This makes SGX unsuitable for large-scale computations. This is mitigated in some newer CPUs with a larger enclave size [56], though this does not help the expense of paging data in/out of enclave memory.

(3) Incomplete standard libraries: While much of the standard C and C++ libraries can be used, some functionality such as input/output, locales, and system calls cannot be used.

(4) Practical vulnerabilities: While the current SGX specification is secure (or at least not yet proven insecure), many practical attacks have been demonstrated [12, 31, 40, 59, 60, 88], damaging the reputation of secure hardware in the research community. Fortunately, these attacks are difficult to execute in practice and can be mitigated. Some attacks are against multithreaded SGX execution [88], so to improve security it may be desirable to run SGX applications with only a single thread, though this may hurt performance.

2.2.1 SGX Efficiency. According to our prior experiments, for the small-scale arithmetic computations that can fit into the capacity of SGX enclaves, the computations inside SGX run 2-3 orders of magnitude faster than the corresponding homomorphic computations. However, simply performing as much subcomputation as possible inside SGX enclaves will result in significant extra overhead. Maximum capacity for the user applications inside the EPC memory area is approximately 96MB, and the overhead increases up to 2000 times [9, 10, 52] when the scale exceeds this capacity because expensive paging and context switches occur [74]. It is nontrivial even for the manufacturer to increase this limit due to the integrity tree overhead and other constraints [74]. Furthermore, SGX is disadvantaged in distributed computing that requires communication due to the overhead of system calls [6, 89]. For the same reasons, SGX incurs significant performance penalties when it loads inputs from a large number of users due to the overhead of context

switching as well as the paging [25, 70]. On the other hand, various optimizations in lattice-based cryptography have greatly improved its efficiency and throughput, making their throughput close to that of plain computations under certain conditions [21, 75]. Distributed computing can be applied to homomorphic computations, especially with special hardware such as GPUs [29, 30, 85], FPGAs [50, 68, 69, 81], and others [67, 78]. However, they significantly outperform computations with SGX under certain conditions (e.g., large-scale inputs, simple operations, small ciphertext parameters), and they can be slower than the corresponding computations inside SGX even with paging under other conditions.

2.2.2 SGX Programming Model. Programming applications for the SGX is nontrivial. An application must be split into trusted and untrusted segments of code, and the programmer must manually specify entry into and exit from the secure enclave (ECALLs and OCALLs). Further, C++ standard library types (e.g., std::vector) and nonstandard classes cannot be passed in ECALLs and OCALLs, forcing programmers to serialize and marshal data in and out of the enclave with buffers. Applications cannot use I/O or system calls from inside the enclave, but must make OCALLs out to untrusted space. Due to these constraints, some expertise is required to program for the SGX, and existing applications cannot be easily ported to the SGX.

3 RELATED WORK

3.1 TEEFHE

One disadvantage of HE is that to achieve Fully Homomorphic Encryption, ciphertexts must be refreshed to mitigate noise added from multiplications. The refreshing procedure, referred to as "bootstrapping", is intensive and complex to implement. As an alternative, a trusted party holding the secret key could decrypt and re-encrypt ciphertexts to create fresh encryptions. TEEFHE [87] used TEEs as the trusted party, thus preventing any disclosure of the secret key outside of a secure environment.

The TEEFHE system offers homomorphic computation as a cloud service. Cloud nodes run homomorphic computations on ciphertexts encrypting user data, and those nodes will have access to TEE-enabled ciphertext refreshment as an oursourced service. TEEFHE uses Microsoft SEAL [72] as its FHE library, and Intel SGX as its TEE. SEAL implemented a Simulator class, which can be used to estimate the remaining noise budget in ciphertexts, and thus decide when outsourced refreshment is necessary. Current publicly available versions of SEAL no longer provide access to the Simulator class, which makes continuations of this research infeasible without access to internal versions of SEAL.

TEEFHE's design is to construct a simple API for its TEE nodes, consisting of operations to configure FHE parameters, securely read in users' secret keys, and refresh ciphertexts. Because SGX is vulnerable to side-channel attacks [17], simply keeping the secret key privately inside the memory enclave is not sufficient to guarantee security. Data-dependent computations may reveal some parts of the secret key under side-channel analysis. To mitigate this, TEEFHE uses code from SEAL that is modified to not exhibit memory accesses, branching, or variable-time computation that is dependent on the secret key.

At 80 bits of security, TEEFHE showed an improvement of two orders of magnitude over SEAL using bootstrapping (another feature not currently publicly available). Notably, side-channel mitigation did not result in any noticeable degradation in performance.

3.2 VISE

The VISE system [25] is another effort to combine TEEs and HE for cloud-based secure computation. In HE, because the values of variables are hidden, branching on encrypted conditions cannot be easily done. Further, large ciphertext expansions may make it infeasible for sensor nodes on slower networks (e.g., satellite Internet) to send homomorphically encrypted data. TEE-only computation with SGX is not suitable for cloud-based computation, due to memory limits and a strict binding between an SGX-based process and its physical host.

VISE solves these problems by using TEEs (Intel SGX, specifically) for both a data gateway and facilitating conditional computations for a cloud homomorphic computation system. Data owners send their data to VISE's TEE servers, using non-homomorphic encryption to reduce communication overhead. The TEE servers homomorphically encrypt user data and forward it to a traditional cloud cluster, which has the advantages of flexibility and scalability according to demand and available resources. As needed, the cloud cluster may return ciphertexts to the TEE servers for secure decryption conditional computation, and reencryption. Final results of batch computations and real-time analytics are also managed by the TEE servers.

VISE uses custom implementations of the DGHV [82] and TFHE [24] schemes. As mentioned in Section 2.1, these schemes operate over single bits or arrays of bits, making them most useful for logical operations but less useful for arithmetic operations. VISE modifies the original source code of implementations of DGHV and TFHE to make them SGX-friendly, involving an SGX port of GMP [41] and custom polynomial arithmetic.

3.3 Other Similar Work

TEEFHE and VISE are the works most similar to ours, in aiming to combine fully homomorphic encryption with SGX for high-performance systems. There is other work that more generally applies SGX and TEEs for improved cryptographic protocols.

Some work uses SGX to guarantee the integrity of homomorphic operations and secure computation. In the system of Drucker and Gueron [33, 34], SGX is used to guarantee the integrity of homomorphic operations in the Pallier additively homomorphic scheme, and (somewhat surprisingly) this system showed a slowdown of less than 2× against solely using SGX or homomorphic encryption. These results are unlikely to be replicated for more complex and expensive fully homomorphic encryption schemes. Kuccuk et al. [51] use SGX to implement a trusted third party for use in secure multiparty computation. The fully homomorphic TFHE scheme was ported to the SGX by Singh [73], and resulted in a slowdown of about 100×, which can be improved upon further.

Other lines of work use an SGX for data management and trusted computing for use with other cryptographic primitives. The Iron system [38] uses SGX to construct functional encryption by having client-hosted SGX processes perform function evaluation and decryption upon authentication from a server SGX. The COVID

contact tracing system of Takeshita et al. [77] uses SGX to manage HE-based Conditional Private Set Intersection, mitigating the scalability issues of SGX by only having the SGX read inputs from infected individuals. Similarly, Wang et al. [84] and Luo et al. [55] apply SGX-based protocol management and secure computing for private auctions and private user matching, respectively. Karl et al. [47] use a TEE to provide noninteractivity for general-purpose secure multiparty computation, and later works of Karl's [48, 49] use SGX to provide noninteractive fault recovery and multivariate polynomial aggregations in private stream aggregation.

Some other work combines SGX and cryptography to improve the performance of trusted computing. The SAFETY and SCOTCH systems [20, 70] use SGX and additive homomorphic encryption for secure queries on patient data, showing an increase over solely using SGX. Use of the partially homomorphic Pallier cryptosystem [63] does not lend post-quantum security or the ability to perform homomorphic operations beyond addition, and SAFETY does not consider query privacy.

4 GPS AND ITS INSTANTIATION

4.1 Models and Assumptions

In previous works [25, 87] constructing systems combining TEEs and fully homomorphic encryption, the use of the SGX is directly dependent upon the number of users. For the relatively small scales evaluated (less than 30 clients for TEEFHE and 1000 for VISE), this is not a serious limitation. However, for computations on much larger scales that are linearly dependent on the number of users, a larger number of users may incur unacceptable overheads on SGX nodes due to the overhead in paging. We thus examine the use of a TEE in scenarios not directly dependent upon the number of users.

System Model. We consider the following parties in this setting.
  (1) Users who provide individual data points to untrusted server. The total number of users may be large.
  (2) An untrusted server running high-performance homomorphic computations to collect and aggregate users' data points to compute certain functions. The server is equipped with SGX, which can be relied upon for lightweight assistance, but is unable to handle computations at the scale of the number of users due to its limits in paging overhead and total enclave space. (This is shown in the experimental results given in Section 1 - the SGX's performance degrades at a much faster rate than that of an HE-based approach.)

In GPS, the SGX generates FHE keys, and allows the public key to be distributed, while keeping the secret key securely in its enclave. Users will homomorphically encrypt their data, and send it to the server for homomorphic computation. The server will run some computations homomorphically, possibly using acceleration techniques such as GPU or other hardware acceleration, or highly parallel computing. As needed, intermediate results are sent into the SGX, decrypted, operated upon, reencrypted, and returned to untrusted memory. Finally, the server sends ciphertexts encrypting the final result, which the SGX will decrypt and return to the server. (Though the name of GPS comes from the 3 technologies used in its implementation, the principles and design of GPS can be applied

with other combinations of secure hardware and homomorphic encryption software.)

While multi-key FHE [64] has some applications in distributed computations among many parties, in this scenario an ordinary FHE scheme suffices. This is because while input comes from a large number of users, the ciphertexts can all be encrypted with respect to a single key pair - the keys of the server, of which the secret key is held by the server's SGX.

4.2 Threat Model and Security

We consider the case of an honest-but-curious adversary who may eavesdrop on user-server communications or compromise any combination of the server and users. A security definition saying that an adversary learns nothing about uncompromised users' inputs is not appropriate, as the server learns the final result of the computation, from which information may be gleaned. In such scenarios where the final result may be released to a compromised party, approaches such as differential privacy [36] should be used to protect user privacy. We only discuss information leakage; total input privacy via differential privacy is an orthogonal issue and dependent on the computation at hand. Instead, we follow a commonly used security definition [76]: no adversary learns more from the real execution of the protocol than in the ideal functionality of the system.

Parties: There exist n users, and one untrusted server with an SGX.
Inputs: Each user provides one input x_i, and a function $f(X)$ on the set $X = \{x_i\}$ of user inputs is publicly known.
Output: The untrusted server learns $f(X)$.

Figure 3: Ideal Functionality of GPS for Secure Computation.

The ideal functionality in this scenario is that all users input their values to a black-box protocol assumed to be secure, and the server receives the result of the computation. This is given in Figure 3, and security according to this functionality is defined in Definition 4.1. (It is easy to also consider different scenarios where users or another party learn the computation's result, though this may require extra communication between the SGX and recipient of the result. We leave this to future exploration.)

Definition 4.1. A protocol Γ securely computes a function $f(\cdots)$ according to the ideal functionality in Figure 3 for a security parameter λ when for all probabilistic polynomial-time (PPT) adversaries $\mathcal{A}$ operating against Γ, there exists a PPT simulator S operating against the ideal functionality δ given in Figure 3 such that for every set of inputs $X = \{x_i\}$, the views of $A_\Gamma(\lambda, X)$ of A in the real protocol and $S_\delta(\lambda, X)$ of S in the ideal protocol are computationally indistinguishable.

For our notion of security, we assume that the SGX is secure against side-channel attacks; such issues are orthogonal to our work. Such vulnerabilities are addressed in other work [18, 45, 61, 62, 86]. Other orthogonal issues such as robustness to user failure are also not considered in this work.

While we only consider honest-but-curious parties, it is worth mentioning one concern that arises in the case of a fully malicious

server. The server could provide the SGX with users' initial inputs instead of a penultimate result from computation awaiting decryption by the SGX. This type of attack is out of scope for our honest-but-curious security model, but is addressed by other work using SGX to guarantee integrity of FHE operations [33, 34].

4.3 Scheme and Library Choice

C and C++ are the languages of choice for SGX programming, so we did not consider implementations in other languages such as Lattigo [58]. The most prominent C++ homomorphic encryption libraries are Microsoft SEAL [72], HElib [43], and PALISADE [65], all of which implement efficient polynomial ring arithmetic using Residue Number System [11, 42] and Number-Theoretic Transform [54] techniques. Of those three, PALISADE implements the most schemes, provides a very wide array of additional functionality such as signatures and identity-based encryption [71], is extensively documented with examples, and is the most actively developed and supported. PALISADE also uses library-level multithreading to improve performance transparently to the user. (SEAL is also highly efficient, but it only implements the B/FV and CKKS schemes, and several useful parts of SEAL are not publicly available, such as bootstrapping and noise estimation.) For these reasons, we use PALISADE in our system, and as a result our choice of library does not limit our choice of scheme for future work.

We choose the CKKS scheme for our implementation due to the usefulness of floating-point arithmetic for applications such as logistic regression and machine learning. CKKS, like BGV and B/FV, allows batching for high throughput, which is desirable for the large-scale computational system we consider. While our implementation uses CKKS, the system can easily run using BGV, B/FV, or TFHE, all of which are supported by PALISADE.

4.4 Approach to Integration

PALISADE and its dependencies cannot be run directly in SGX, as SGX applications can use only a limited set of the C and C++ standard libraries (as discussed in Section 2.2.2). Functionality such as input/output and system calls is disabled, and some functions are not reimplemented in the SGX SDK's provided C/C++ libraries. These restrictions have forced prior work to partially or fully reimplement homomorphic encryption libraries for use in SGX applications [25, 87]. While this approach is possible, it is not desirable for developers to need to rewrite libraries whenever a particular library or scheme is desired for use with SGX, especially considering the high complexity of homomorphic encryption and the more sophisticated libraries implementing it (e.g., PALISADE, SEAL).

A more preferable approach is to utilize a general and reusable method of porting FHE libraries to SGX. One such method is the use of the Graphene containerization framework [79, 80]. Graphene is a library OS, which exposes needed library and system functionality through its own shared libraries to user applications, and itself runs on the host OS. Graphene has already been shown to be a viable method of porting applications and libraries originally written for ordinary environments to SGX [53]. By using Graphene, we can easily run unmodified PALISADE applications in SGX, without any need to modify PALISADE. Some minor configuration is needed for passing arguments and environment variables, specifying allowed input/output files, and importing PALISADE shared libraries as trusted files.

As mentioned in Section 2.2, multithreading in SGX may expose side-channel vulnerabilities [88]. To mitigate this, we disable multithreaded execution. By default, PALISADE uses OpenMP to provide multithreading in homomorphic operations. For the SGX application, we disabled this functionality by default. A rewrite of PALISADE (a very large and full-featured library) to remove all side-channel attacks, as in TEEFHE [87], is orthogonal to this project. Also, multithreading in SGX may incur overhead due to the use of kernel threads to back application-level threads [52, 80].

This combination of technology - Graphene, PALISADE, and SGX - is our system, GPS, that we can apply to computations not easily handled by SGX or HE alone.

4.4.1 Security (Informal). GPS is secure according to Definition 4.1. It is easy to see that security of user inputs and intermediate computations prior to the final result is preserved. By the semantic security of homomorphic encryption, no adversary learns any new information about uncompromised users' ciphertexts sent to the server, and the server cannot learn anything about the data on which it is operating. Data operated upon by the SGX is encrypted securely in enclave memory, and is thus also protected by semantic security (assuming no side-channel vulnerabilities). Thus no adversary can learn from their view any information about user data that would not be learned in the ideal case from seeing compromised users' inputs and the server's result, showing security.

4.5 Methodologies/Strategies of Integration

When adapting a computation for use with GPS, a careful consideration of the specific traits of the computation along with the relative advantages and disadvantages of SGX and HE. We recall that HE is highly parallelizeable and scalable, but may be limited by some factors such as multiplicative depth or complex calculations not easily expressible in arithmetic circuits. In contrast, SGX is not limited by multiplicative depth or complexity, being capable of arbitrary computations, but the SGX is not well-suited for dealing with many different inputs from many different users, due to memory space limits and paging overhead.

Thus to intelligently apply GPS, portions of the computation that are directly polynomially dependent on the number of users/inputs (i.e., $\Omega(n)$ for n users) in computation or memory should be run outside the SGX, by the untrusted server, who can bring much more memory and scalability to bear. On the other hand, computations that are $O(1)$ but difficult to compute homomorphically (i.e., requiring conditionals, high multiplicative depth, or operations not easily approximated with only addition and multiplication) can be performed more easily in the SGX. For similar reasons, the size of the final result should be small and not dependent on the number of users. A given computation can be described in terms of its abstract dataflow (e.g., as a directed acyclic graph, though such a formal description may not be necessary), and each stage of the computation should be analyzed to determine whether it is better suited to HE or SGX. This choice is a heuristic one, based on factors such as the computation's multiplicative depth, the size of inputs to the stage of the computation, and the feasibility of implementing the computation within the restrictions of HE (i.e. strict arithmetic

circuits without looping or conditionals). To deal with the accumulation of multiplicative depth, SGX-based bootstrapping stages can be inserted as needed in the computation, similarly to TEEFHE [87].

This formulation results in a protocol informally described as follows:

(1) Preprocessing: Given a computation to run, the computation is divided into stages as described above, with each of the SGX and untrusted server being assigned the stages that are better suited to their strengths.

(2) Setup: The SGX generates FHE keys, and distributes public keys to users and evaluation keys (e.g., relinearization keys) to the server.

(3) User Input: Users encrypt their data under the provided public key, and send their encrypted data to the server.

(4) Computation: The server and SGX perform the divided computation, passing data between them as needed. The SGX can decrypt data it receives from the server, and reencrypt before passing intermediate results back to the server.

(5) Final Result: The SGX receives the penultimate result (homomorphically encrypted) from the server, if the final stage of the computation was not in the SGX. Then, the SGX decrypts the result, and returns it to the server.

It should be noted that GPS is a protocol that utilizes function-dependent HE, and thus all the regular difficulties of writing HE applications apply. Issues such as parameter selection, optimization of the computation for multiplicative depth and minimal computation, and deferment of relinearization still must be considered by HE experts.

4.6 Instantiation: Linear Regression Training

As an instantiation of our idea to be used in evaluation, we chose the training of linear regression models. Note that the focus of this paper is not on the optimization of linear regression training; this instantiation is an example to show concrete ideas and experiment results. Therefore, we only focus on how much improvement we gain by applying GPS to this task, rather than whether the OLS is the best option for the linear regression training. A more full investigation of varied securely computed functions is a topic for future work.

Linear regression is a simple method of attempting to choose weights in a model. For n users, each with a vector of p observed independent variables $\mathbf{x}_i \in \mathbb{R}^p$ and a dependent variable $y_i \in \mathbb{R}$, the Ordinary Least Squares (OLS) method for training a linear regression model computes

$$\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

where $\mathbf{X}$ is the $n \times p$ matrix whose n rows consist of the vectors $\mathbf{x}_i$, and $\mathbf{y}$ is the vector of length n whose entries are the users' responses y_i. While approximations, other regression methods, and other types of multi-user secure computation exist, we chose OLS for this work for its simplicity and wide applicability.

Linear regression is a good application for combining homomorphic encryption with trusted computing for the following reasons:

(1) It can take advantage of light trusted computing assistance at some stages such as inverse computation, which are difficult to do with purely homomorphic computation.

(2) The larger portions of the computation scales linearly with the number of users, making it a difficult task at scale for an SGX - for example, computing $\mathbf{X}^T\mathbf{X}$ is multiplying $p \times n$ and $n \times p$ matrices.

(3) The size of smaller computations (e.g., finding the inverse of the $p \times p$ matrix $\mathbf{X}^T\mathbf{X}$) and the final result are not dependent on the number of users.

Concretely, all n users will send their p homomorphically encrypted inputs $\mathbf{x}_i, y_i$ to the server. The server will combine its received inputs for batched evaluation (see *Batching for Linear Regression* below) as $\mathbf{X}$, $\mathbf{X}^T$, and $\mathbf{y}$, and will then first compute $\mathbf{X}^T\mathbf{X}$. That result is only a $p \times p$ matrix, and as p is a small number not dependent on the number of users (usually, $p < 20$), the SGX can easily handle computing its inverse. In parallel, $\mathbf{X}^T\mathbf{y}$ is also computed homomorphically, resulting in a $p \times 1$ vector. Next, $\mathbf{X}^T\mathbf{X}$ and $\mathbf{X}^T\mathbf{y}$ are sent into the enclave and decrypted. The elements packed in each ciphertext are summed up to complete the dot product (see below). The SGX then computes the matrix inverse $(\mathbf{X}^T\mathbf{X})^{-1}$. Finally, the product $\beta = (\mathbf{X}^T\mathbf{X})^{-1}(\mathbf{X}^T\mathbf{y})$ is computed, resulting in a $p \times 1$ result, which is then returned to the server.

Batching for Linear Regression. Previous work in homomorphic regression training used batching to pack thousands of operands into a single ciphertext [13, 14]. Because the scale of computation we consider is much larger than in previous work, we cannot directly apply previously-used batching techniques emplacing entire matrices into single ciphertexts. Further, the structure of OLS linear regression does not allow any training to occur without input from multiple users. This is because the rows of $\mathbf{X}^T$ (equivalently, the columns of $\mathbf{X}$) are comprised of values from each of the n users, and n may be too large to use ordinary methods of packing all users' information into a single ciphertext. However, we can still apply batching to improve throughput.

Suppose we can batch B operands in a single ciphertext. (In CKKS, B is equal to $N/2$, where N is the polynomial modulus degree; $B = N$ in BGV and B/FV.) We can use batching to reduce the number of homomorphic multiplications by a factor of B, at the cost of $n \cdot p$ additions. In our application, we can have all n users u_i send p ciphertexts $x_{i,j}$ for $i \in [0, N)$ and $j \in [0, p)$. These ciphertexts encrypt user i's j^{th} regressor. Users will also send ciphertexts y_i for $i \in [0, n)$ encrypting their response value. All ciphertexts have the actual data encrypted at slot $i \pmod N$, with all other slots zero-valued. Then to pack the values $x_{i,j}$ into ciphertexts, the server can simply homomorphically add users' ciphertexts.

The server utilizes packing to compress the rows of $\mathbf{X}^T$, and equivalently the columns of $\mathbf{X}$, and similarly to reduce the size of $\mathbf{y}$. Doing this reduces the number of operands in a row or column from n to $\lceil \frac{n}{B} \rceil$, reducing the number of both additions and multiplications that must occur by a factor of roughly B. When packed ciphertexts are decrypted by the SGX, all B of their elements must be additively aggregated into a single value.

Normally, the multiplicative depth of a computation is the determining factor in the computations a SHE scheme can compute. This is because homomorphic multiplications increase ciphertext noise multiplicatively, while homomorphic additions only increase noise additively. While the multiplicative depth required for a matrix multiplication is technically only 1 in both cases, for values of

n in the millions, the noise gained from the n additions becomes non-negligible. By using this manner of batching, we can reduce the number of homomorphic operations, thus allowing us to use smaller parameters, which reduces ciphertext size and homomorphic operation runtime.

5 EXPERIMENTAL EVALUATION

5.1 Implementation

Our test workstation used an Intel Xeon CPU with 20 cores operating at 3.7GHz and 128GB memory, using Ubuntu 18.04. Our tests, written in C++, used the Development version of PALISADE, version 1.11.2. PALISADE uses OpenMP [28] for multithreading its operations, and we also used OpenMP to parallelize matrix multiplication. We conducted experiments with threads both enabled and disabled. In our hybrid system, relinearization was not used for homomorphic multiplications, significantly improving our runtime.

We wrote PALISADE programs for the pack-and-multiply used to calculate $\mathbf{X}^T\mathbf{X}$ in linear regression, as well as a general matrix multiplication program. Using Graphene and PALISADE, we wrote SGX programs to calculate a matrix inverse, perform a final multiplication, and decrypt the final output. An SGX-only implementation of multi-user linear regression was also implemented as a baseline. We time each portion of the computation, and report the results. We also implemented logistic regression using only homomorphic encryption, but quickly realized that for our scale, this strategy is not feasible, due to the high complexity of computing a determinant and matrix inverse. This further supports our decision to run small-scale yet difficult operations inside the SGX. Our source code is available at https://anonymous.4open.science/r/LinReg_SGX-A73D/ and https://anonymous.4open.science/r/integrated-linear-regression-81D7/

5.2 Experimental Results

We evaluated scenarios with varying values of p and n. We allow p to vary from 2 to 12, and n to vary from $1,000,000$ to $10,000,000$, in increments of $1,000,000$. All results shown are the average of 10 trials, except in the SGX implementation with $n = 5,000,000$, $p = 12$, which ran for 5 trials due to the high runtime. Standard deviations for results were within $\pm2.2\%$ of the mean, and $\pm1\%$ for the single-threaded comparisons to SGX. We describe improvement in terms of speedup, the ration of the baseline's runtime to the new system's runtime.

Setting a polynomial modulus degree of $N = 8192$ gives us $B = 4096$ packed elements per CKKS plaintext. Due to our batching, we need only perform dot products on vectors of ciphertexts of length $\lceil\frac{n}{B}\rceil \leq 2442$. Each element was created from $B - 1 = 4095$ homomorphic additions, so the total number of homomorphic additions is no more than 6537, with only one homomorphic multiplication needed for matrix multiplication. This allows noise bounds tolerant enough for us to only need two ciphertext moduli totalling 111 bits, giving this instantiation of CKKS between 192 and 256 bits of (classical) security - greater than the 80 bits used in TEEFHE [87], and the 128 bits evaluated by VISE [25] (though VISE usually used 64 bits of security for performance reasons). SGX's AES encryption provides at least 128 bits of security, so from this and our CKKS parameters we can conclude that our setting has at least 128

bits of security. These parameters result in ciphertext sizes of approximately only 265KB, keeping users' communication overhead small.

5.2.1 Single-Threaded Comparison to SGX. We first evaluate our performance by comparing the performance of a basic GPS implementation of linear regression directly to an SGX-only implementation of linear regression. As shown in Figure 5 and table 2, while both systems' runtime scales linearly with the input size, GPS shows better performance and scaling as the number of inputs increases, achieving speedups from 1.13× to 2.07×. It should be noted that this comparison is for a large-scale multi-user computation; for smaller workloads SGX is likely to outperform GPS. Figure 4 also shows how GPS' performance is also better than SGX as the number of dependent variables increases. As shown in Table 1, GPS averages a 1.16× to 8.69× speedup. These experiments show that on a large scale, a basic implementation of GPS without optimizations such as hardware acceleration, parallelized matrix operations, etc. outperforms SGX, though both exhibit the same asymptotic behavior.
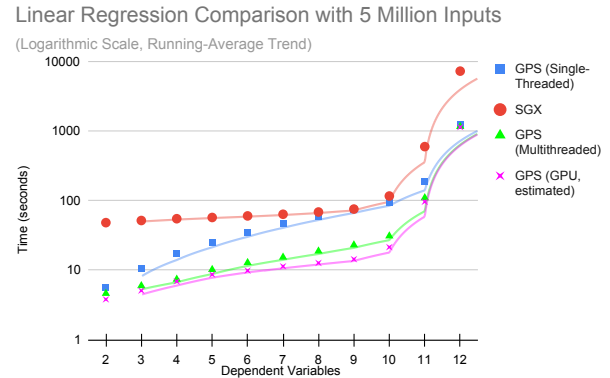

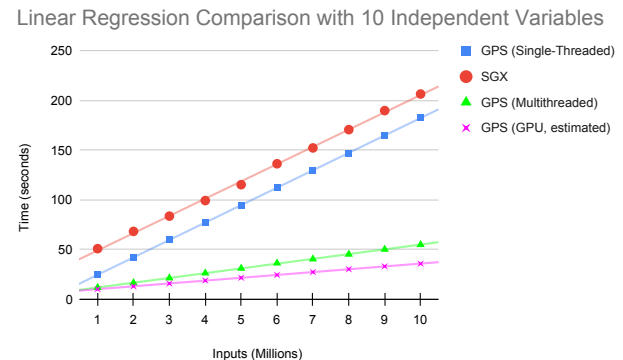**Figure 4: Linear Regression Comparison of GPS and SGX with $n = 5,000,000$**

Figure 5: Linear Regression Comparison of GPS and SGX with $p = 10$

Jonathan Takeshita, Colin McKechney, Justin Pajak, Antonis Papadimitriou, Ryan Karl, and Taeho Jung

5.2.2 Multithreaded Performance. Our evaluation in Section 5.2.1 used only a single thread for each program (though the first two matrix multiplications were computed in parallel). Using OpenMP [28], we next parallelized our implementation of matrix multiplication using PALISADE, in order to exploit one of GPS' advantages: the ability to parallelize homomorphic computation. We used 18 threads in total, and found that the optimal division of threads was to give 10 to the calculation of $\mathbf{X}^T\mathbf{X}$ and 8 to the calculation of $\mathbf{X}^T\mathbf{y}$. Figures 4 and 5 show the performance achieved by applying PALISADE's multithreading to GPS, and Tables 1 and 2 show the improvements that this brings as compared to single-threaded SGX. While consistent speedups are shown, the smaller improvements indicate that the homomorphic operations parallelized by PALISADE were not major bottlenecks. Overall, multithreaded GPS achieved speedups from 2.09× to 3.32× for increasing n and from 1.08× to 3.14× for increasing p over single-threaded GPS. Against SGX, multithreaded GPS showed improvements from 2.09× to 3.32× for increasing n and from 3.28× to 10.43 for increasing p. Future work in applying parallelism at a higher level (e.g., parallelizing matrix arithmetic) may be able to further improve this.

5.2.3 Estimation of Speedup from GPU Acceleration. Due to the computational and memory demands of HE, much research has been undertaken into hardware acceleration of HE. Prior work has examined the use of GPUs, FPGAs, ASICs, and other hardware solutions [7, 29, 30, 57, 67–69, 78, 85]. The most widely available and used of these technologies is GPU. A recent GPU implementation of CKKS showed speedups of one to two orders of magnitude against Microsoft SEAL [7]. In particular, speedups of about 25× were reported for benchmarks of homomorphic addition and multiplication at $N = 8192$, and 20× speedups were reported for real-world inference computations. Taking a conservative estimate of 20× speedup for our matrix multiplication (which is comprised entirely of additions and multiplications), we then estimate the improvement we can gain from GPU acceleration of matrix multiplication, and show the results in Figures 4 and 5 and tables 1 and 2. (These tests did not count runtime for process startup/cleanup, which is small.)

The 20× speedup in matrix multiplication time translated to estimated speedups of only 4.99× to 5.77× for increasing n and 5.25× to 12.67 for increasing p against SGX. This suggests that while hardware acceleration of homomorphic operations is useful for improving the protocol's latency, the main obstacle of the computations that GPS is best applied to is the overhead marshalling inputs from a large number of users.

5.3 Analysis
This initial investigation of GPS generally shows consistent improvement over SGX-only implementations, which can translate to a large reduction in latency for large-scale computations with data from a large number of users. Having designed and implemented GPS, future work can now focus on attaining orders-of-magnitude improvements.

Techniques such as multithreading and GPU acceleration show or project an improvement of up to approximately 20× for the arithmetic portions of GPS. However, applying or projecting these shows only improvements of 2.09× to 12.67×. This indicates that the bottleneck in our implementation of GPS is mainly due to the

overhead from other intensive portions of our computation, such as matrix inversion and input/output.

Interestingly, matrix inversion was slower in the SGX-only version than in GPS. This is most likely due to the additional memory consumption of the SGX-only version, which had previously read in all $n \times p$ user inputs. That memory use increases the amount of paging that needs to take place, and memory paging is a slow operation for the SGX due to the need for memory encryption/decryption. This further reinforces the efficacy of our design in witholding the full set of user inputs from the SGX.

6 LIMITATIONS
Our integration of SGX and SHE for the first time addresses the performance drawbacks of SGX in large-scale distributed data aggregation, which resulted from the loading of a large number of inputs from users into SGX enclaves that causes frequent system calls and context switches. The integration is not effective in the cases where the loading of user inputs does not cause frequent system calls (e.g., the number of users is small, the total size of aggregated datasets is small).

In many cases, SGX-only computation will remain a better choice, particularly in scenarios unlike ours, with a limited number of inputs and little opportunity for parallel processing. Overall, the basic use of GPS can improve the runtime of computations, but does not improve the asymptotic runtime or scalability of a computation without the use of other techniques (e.g., parallelization). Also, not all computations are most advantageously adapted to GPS: computations without a large number of inputs and reducing the size of intermediate computations may be better suited for more traditional approaches of applying SGX or HE.

The experimental results shown in Section 5.1 show only modest speedups; while this does show an improvement over SGX, this suggests that GPS can still be improved to maturity by exploring further integration with parallelization and hardware acceleration.

GPS does not address some pertinent obstacles to widespread adoption of HE and secure computing. GPS requires expert knowledge of both SGX and HE to implement correctly, which may discourage its use by non-experts. Common obstacles such as parameter selection for performance and security, choices of scheme and data encoding, and programming in the HE paradigm are still present in GPS. Fortunately, due to our novel utilization of Graphene for the SGX portions of GPS, developers do not need to have expertise in the particulars of SGX programming as needed for development with the SGX SDK; however, programmers still need to be acutely aware of the strengths and limitations of SGX to use GPS effectively. Future use of FHE transpilers [2, 32, 83] may further reduce the barriers to entry in developing for HE and GPS.

7 CONCLUSION
In this paper, we showed the potential of combining homomorphic encryption and trusted execution for large-scale multi-user computations not suited to HE-only or SGX-only computation. Further, we pioneered the use of containerization to use existing HE libraries in SGX without needing to modify the libraries or our application. Our experimental results show the improvements of GPS over an SGX-only implementation of linear regression; speedups of 1.13× to 2.07× as the number of dependent variables scale and 1.03× to

Table 1: Speedup for GPS and Optimizations with Increasing p ($n = 5,000,000$)

Dependent Vars.	2	3	4	5	6	7	8	9	10	11	12
GPS vs. SGX	8.69	4.84	3.18	2.26	1.73	1.38	1.16	1.03	1.23	3.21	5.80
Multithreaded GPS vs. SGX	10.43	8.67	7.43	5.61	4.70	4.15	3.65	3.28	3.76	5.45	6.27
Multithreaded GPS vs. GPS	1.20	1.79	2.34	2.48	2.72	3.01	3.14	3.20	3.07	1.69	1.08
GPU-Accelerated GPS (estimated) vs. SGX	12.67	10.18	7.96	6.60	6.10	5.60	5.40	5.25	5.43	6.20	6.38
GPU-Accelerated GPS (estimated) vs. GPS	1.46	2.10	2.50	2.92	3.54	4.06	4.64	5.12	4.43	1.93	1.10

Table 2: Speedup for GPS and Optimizations with Increasing n ($p = 10$)

Inputs (Millions)	1	2	3	4	5	6	7	8	9	10
GPS vs. SGX	2.07	1.62	1.40	1.28	1.22	1.21	1.18	1.16	1.15	1.13
Multithreaded GPS vs. SGX	4.33	4.12	3.91	3.77	3.71	3.74	3.76	3.77	3.78	3.76
Multithreaded GPS vs. GPS	2.09	2.54	2.79	2.94	3.04	3.09	3.19	3.25	3.29	3.32
GPU-Accelerated GPS (estimated) vs. SGX	4.99	5.24	5.27	5.25	5.33	5.56	5.56	5.63	5.72	5.77
GPU-Accelerated GPS (estimated) vs. GPS	2.41	3.23	3.76	4.09	4.37	4.60	4.72	4.85	4.98	5.10

8.69× as the number of users scale. When applying PALISADE's multithreading capabilities, we can increase the improvement over SGX to 3.71× to 4.33× with increasing n and 3.28× to 10.43× over SGX. We project that this improvement can also be increased by GPU acceleration of arithmetic HE computations, for speedups of 4.99× to 5.77× for increasing n and 5.40× to 12.67× for increasing p. Besides the improvements shown, our results indicate bottlenecks due to high I/O overhead from many different users, which informs our priorities for future research.

There are many possible directions for future work. One avenue is in investigating the combination of SGX and other hardware platforms. Another is in examining more varied applications, such as statistical calculation and machine learning tasks. An obstacle encountered in this work was the high overhead from I/O of user input; future work examining how to mitigate this will be able to have a great impact on the latency of such integrated systems. While some work exists in combining lightweight, purpose-built cryptographic protocols with SGX (Section 3.3), the principles of GPS can be further applied for such integrations for even better efficiency and functionality. Another exciting new direction is in the use of FHE transpilers [2, 32, 83]. As noted in Section 4.5, expertise with HE is still needed to apply GPS. Use of a transpiler to programmatically convert source code from plain operations to encrypted computation can allow non-experts to implement their computations securely with HE libraries. Future work in transpilers could even examine specifically targeting GPS. GPS has shown improvement over the basic case of SGX-only computation, and has many potential avenues for further improvement, which we hope to explore in future work.

REFERENCES

[1] [n.d.]. 37 Mind Blowing YouTube Facts, Figures and Statistics – 2021. https://incomelords.com/youtube-statistics/. Accessed: 2021-09-26.

[2] [n.d.]. A General Purpose Transpiler for Fully Homomorphic Encryption. ([n. d.]).

[3] [n.d.]. Intel product specifications. https://ark.intel.com/content/www/us/en/ark/search/featurefilter.html?productType=873&2_SoftwareGuardExtensions=Yes+with+Intel%C2%AE+SPS. https://ark.intel.com/content/www/us/en/ark/search/featurefilter.html?productType=873&2_SoftwareGuardExtensions=Yes+with+Intel%C2%AE+SPS

[4] [n.d.]. Number of daily active Facebook users worldwide as of 2nd quarter 2021. hhttps://www.statista.com/statistics/346167/facebook-global-dau/. Accessed: 2021-09-26.

[5] 2021. 2021 Privacy Enhancing Technologies request for proposals. https://bit.ly/3b8DFf0

[6] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX.. In *NDSS*.

[7] Ahmad Al Badawi et al. 2020. PRIVFT: Private and fast text classification with homomorphic encryption. *IEEE Access* 8 (2020), 226544–226556.

[8] Prabhanjan Ananth, Abhishek Jain, Zhengzhong Jin, and Giulio Malavolta. 2020. Multi-key fully-homomorphic encryption in the plain model. In *Theory of Cryptography Conference*. Springer, 28–57.

[9] Sergei Arnautov et al. 2016. {SCONE}: Secure linux containers with intel {SGX}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 689–703.

[10] Maurice Bailleu et al. 2019. {SPEICHER}: Securing lsm-based key-value stores using shielded execution. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 173–190.

[11] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. 2016. A full RNS variant of FV like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*. Springer, 423–442.

[12] Andrea Biondo et al. 2018. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel {SGX}. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1213–1227.

[13] Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, and Shafi Goldwasser. 2020. Secure large-scale genome-wide association studies using homomorphic encryption. *Proceedings of the National Academy of Sciences* 117, 21 (2020), 11608–11613.

[14] Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, Kurt Rohloff, and Vinod Vaikuntanathan. 2020. Optimized homomorphic encryption solution for secure genome-wide association studies. *BMC Medical Genomics* 13, 7 (2020), 1–13.

[15] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[16] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)* 6, 3 (2014), 1–36.

[17] Ferdinand Brasser et al. 2017. Software grand exposure:{SGX} cache attacks are practical. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*.

[18] Ferdinand Brasser et al. 2019. DR. SGX: Automated and adjustable side-channel protection for SGX using data location randomization. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 788–800.

[19] Long Cheng, Fang Liu, and Danfeng Yao. 2017. Enterprise data breach: causes, challenges, prevention, and future directions. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7, 5 (2017), e1211.

[20] Wang Chenghong et al. 2017. SCOTCH: Secure Counting Of encrypTed genomiC data using a Hybrid approach. In *AMIA Annual Symposium Proceedings*, Vol. 2017. American Medical Informatics Association, 1744.

[21] Jung Hee Cheon et al. 2018. A full RNS variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*. Springer, 347–368.

[22] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 409–437.

[23] Ilaria Chillotti et al. 2016. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *international conference on the theory and application of cryptology and information security*. Springer, 3–33.

[24] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. August 2016. TFHE: Fast Fully Homomorphic Encryption Library. https://tfhe.github.io/tfhe/.

[25] Luigi Coppolino, Salvatore D'Antonio, Valerio Formicola, Giovanni Mazzeo, and Luigi Romano. 2021. VISE: Combining Intel SGX and Homomorphic Encryption for Cloud Industrial Control Systems. *IEEE Trans. Comput.* 70, 5 (2021), 711–724. https://doi.org/10.1109/TC.2020.2995638

[26] Jean-Sebastien Coron et al. 2011. Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers. Cryptology ePrint Archive, Report 2011/440. https://eprint.iacr.org/2011/440.

[27] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016), 1–118.

[28] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5,

1 (1998), 46–55.

[29] Wei Dai, Yarkın Doröz, and Berk Sunar. 2014. Accelerating NTRU based homomorphic encryption using GPUs. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[30] Wei Dai and Berk Sunar. 2015. cuHE: A homomorphic encryption accelerator library. In *International Conference on Cryptography and Information Security in the Balkans*. Springer, 169–186.

[31] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. 2018. Cachequote: Efficiently recovering long-term secrets of SGX EPID via cache attacks. (2018).

[32] Roshan Dathathri et al. 2020. EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Jun 2020). https://doi.org/10.1145/3385412.3386023

[33] Nir Drucker and Shay Gueron. 2017. Combining Homomorphic Encryption with Trusted Execution Environment: A Demonstration with Paillier Encryption and SGX. In *Proceedings of the 2017 International Workshop on Managing Insider Security Threats*. 85–88.

[34] Nir Drucker and Shay Gueron. 2018. Achieving trustworthy Homomorphic Encryption by combining it with a Trusted Execution Environment. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.* 9, 1 (2018), 86–99.

[35] Léo Ducas and Daniele Micciancio. 2015. FHEW: bootstrapping homomorphic encryption in less than a second. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 617–640.

[36] Cynthia Dwork. 2008. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation*. Springer, 1–19.

[37] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* 2012 (2012), 144.

[38] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. 2017. Iron: functional encryption using Intel SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 765–782.

[39] Anders T Gjerdrum, Robert Pettersen, Håvard D Johansen, and Dag Johansen. 2017. Performance of Trusted Computing in Cloud Infrastructures with Intel SGX.. In *CLOSER*. 668–675.

[40] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.

[41] Torbjörn Granlund. 1996. GNU MP. *The GNU Multiple Precision Arithmetic Library* 2, 2 (1996).

[42] Shai Halevi, Yuriy Polyakov, and Victor Shoup. 2019. An improved RNS variant of the BFV homomorphic encryption scheme. In *Cryptographers' Track at the RSA Conference*. Springer, 83–105.

[43] Shai Halevi and Victor Shoup. 2014. Helib. *Retrieved from HELib: https://github.com. shaih/HELib* (2014).

[44] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2019. Logistic regression on homomorphic encrypted data at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 9466–9471.

[45] Zhou Hongwei, Ke Zhipeng, Zhang Yuchen, Wu Dangyang, and Yuan Jinhui. 2021. TSGX: Defeating SGX Side Channel Attack with Support of TPM. In *2021 Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*. IEEE, 192–196.

[46] Intel Corporation 2021. *Intel Architecture Memory Encryption Technologies*. Intel Corporation. Rev. 1.3.

[47] Ryan Karl, Timothy Burchfield, Jonathan Takeshita, and Taeho Jung. 2019. Non-interactive MPC with trusted hardware secure against residual function attacks. In *International Conference on Security and Privacy in Communication Systems*. Springer, 425–439.

[48] Ryan Karl, Jonathan Takeshita, and Taeho Jung. 2021. Cryptonite: A Framework for Flexible Time-Series Secure Aggregation with Online Fault Tolerance. In *17th International Conference on Security and Privacy in Communication Networks*.

[49] Ryan Karl, Jonathan Takeshita, Alamin Mohammed, Aaron Striegel, and Taeho Jung. 2021. Cryptonomial: A Framework for Private Time-Series Polynomial Calculations. In *17th International Conference on Security and Privacy in Communication Networks*.

[50] Sunwoong Kim et al. 2020. Hardware architecture of a number theoretic transform for a bootstrappable RNS-based homomorphic encryption scheme. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 56–64.

[51] Kubilay Ahmet Küçük, Andrew Paverd, Andrew Martin, N Asokan, Andrew Simpson, and Robin Ankele. 2016. Exploring the use of Intel SGX for secure many-party applications. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*. 1–6.

[52] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2019. TensorSCONE: a secure TensorFlow framework using Intel SGX. *arXiv preprint arXiv:1902.04413* (2019).

[53] Dmitrii Kuvaiskii, Somnath Chakrabarti, and Mona Vij. 2018. Snort intrusion detection system with Intel Software Guard eXtension (Intel SGX). *arXiv preprint arXiv:1802.00508* (2018).

[54] Patrick Longa and Michael Naehrig. 2016. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *International Conference on Cryptology and Network Security*. Springer, 124–139.

[55] Junwei Luo, Xuechao Yang, and Xun Yi. 2020. SGX-based Users Matching with Privacy Protection. In *Proceedings of the Australasian Computer Science Week Multiconference*. 1–9.

[56] Dylan Martin. 2020. Intel Xeon Ice Lake CPUs To Get SGX With Expanded Security Features. https://www.crn.com/news/components-peripherals/intel-xeon-ice-lake-cpus-to-get-sgx-with-expanded-security-features

[57] Toufique Morshed, Md Momin Al Aziz, and Noman Mohammed. 2020. CPU and GPU accelerated fully homomorphic encryption. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 142–153.

[58] Christian Mouchet et al. 2020. Lattigo: A multiparty homomorphic encryption library in Go.

[59] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1466–1482.

[60] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. 2020. A survey of published attacks on intel SGX. *arXiv preprint arXiv:2006.13598* (2020).

[61] Hyunyoung Oh et al. 2020. TRUSTORE: Side-Channel Resistant Storage for SGX using Intel Hybrid CPU-FPGA. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1903–1918.

[62] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 {Usenix} Annual Technical Conference ({USENIX} {ATC} 18)*. 227–240.

[63] Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*. Springer, 223–238.

[64] Chris Peikert and Sina Shiehian. 2016. Multi-key FHE from LWE, revisited. In *Theory of Cryptography Conference*. Springer, 217–238.

[65] Yuriy Polyakov, Kurt Rohloff, and Gerard W Ryan. 2017. PALISADE lattice cryptography library user manual. *Cybersecurity Research Center, New Jersey Institute of Technology (NJIT), Tech. Rep* 15 (2017).

[66] Meg Prater. [n.d.]. 25 Google Search Statistics to Bookmark ASAP. https://blog.hubspot.com/marketing/google-search-statistics. Accessed: 2021-09-26.

[67] Dayane Reis, Jonathan Takeshita, Taeho Jung, Michael Niemier, and Xiaobo Sharon Hu. 2020. Computing-in-Memory for Performance and Energy-Efficient Homomorphic Encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 11 (2020), 2300–2313.

[68] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1295–1309.

[69] Sujoy Sinha Roy et al. 2019. FPGA-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE HPCA*. IEEE, 387–398.

[70] Md Nazmus Sadat et al. 2018. Safety: secure gwas in federated environment through a hybrid solution. *IEEE/ACM trans. on computational biology and bioinformatics* 16, 1 (2018), 93–102.

[71] Amit Sahai and Brent Waters. 2005. Fuzzy identity-based encryption. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 457–473.

[72] SEAL 2018. Microsoft SEAL (release 3.0). http://sealcrypto.org. Microsoft Research, Redmond, WA.

[73] Isak Sunde Singh. 2020. *Safe and secure outsourced computing with fully homomorphic encryption and trusted execution environments*. Master's thesis. UiT Norges arktiske universitet.

[74] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramanian. 2018. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 665–678.

[75] Jonathan Takeshita, Ryan Karl, Ting Gong, and Taeho Jung. 2020. SLAP: Simple Lattice-Based Private Stream Aggregation Protocol. Cryptology ePrint Archive, Report 2020/1611. (2020). https://eprint.iacr.org/2020/1611.

[76] Jonathan Takeshita, Ryan Karl, and Taeho Jung. 2020. Secure Single-Server Nearly-Identical Image Deduplication. In *IoTSPT-ML at ICCCN 2020*. IEEE.

[77] Jonathan Takeshita, Ryan Karl, Alamin Mohammed, Aaron Striegel, and Taeho Jung. 2021. Provably Secure Contact Tracing with Conditional Private Set Intersection. In *17th International Conference on Security and Privacy in Communication Networks*.

[78] Jonathan Takeshita, Dayane Reis, Ting Gong, Michael Niemier, X. Sharon Hu, and Taeho Jung. 2020. Algorithmic Acceleration of B/FV-like Somewhat Homomorphic Encryption for Compute-Enabled RAM. In *International Conference on Selected Areas in Cryptography*. Springer.

[79] Chia-Che Tsai et al. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

[80] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A practical library {OS} for unmodified applications on {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 645–658.

[81] Furkan Turan, Sujoy Sinha Roy, and Ingrid Verbauwhede. 2020. HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA. *IEEE Trans. Comput.* 69, 8 (2020), 1185–1196.

[82] Marten Van Dijk et al. 2010. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 24–43.

[83] Alexander Viand and Hossein Shafagh. 2018. Marble: Making fully homomorphic encryption accessible to all. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 49–60.

[84] Jiaqi Wang, Ning Lu, Qingfeng Cheng, Lu Zhou, and Wenbo Shi. 2020. A secure spectrum auction scheme without the trusted party based on the smart contract. *Digital Communications and Networks* (2020).

[85] Wei Wang et al. 2012. Accelerating fully homomorphic encryption using GPU. In *2012 IEEE conference on high performance extreme computing*. IEEE, 1–5.

[86] Wenhao Wang et al. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2421–2434.

[87] Wenhao Wang et al. 2019. Toward Scalable Fully Homomorphic Encryption Through Light Trusted Computing Assistance. *arXiv preprint arXiv:1905.07766* (2019).

[88] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. Async-Shock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security*. Springer, 440–457.

[89] Ofir Weisse et al. 2017. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 81–93.