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

Jonathan Takeshita[1], Colin McKechney[1], Justin Pajak[1], Antonis Papadimitriou[2], Ryan Karl[1], and Taeho Jung[1]

[1] University of Notre Dame, Notre Dame, IN 46556, USA
[jtakeshi,cmckechn,jpajak,rkarl,tjung]@nd.edu
[2] Duality Technologies, Newark, NJ 07103, USA antonis.papadimitriou@gmail.com

Abstract. Secure computing methods such as 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. 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 12.67x with GPU acceleration.

Keywords: Lattice-based Cryptography· Intel SGX· Large-scale Computing

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, while utilizing that data. Gathering users' unencrypted data at datacenters are vulnerable to data breaches, as shown by recent accidents [15]. An alternative is to collect encrypted data from users to avoid such risks.

The modern era presents new challenges in securely processing user data. The first is the scale of the computation. Google performs 5.6 billion searches per day [56] and serves nearly 5 billion videos on YouTube daily [1], while Facebook claims approximately 1.9 billion daily active users [3]. 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. 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 Multiparty Computation (MPC). HE allows computation to take place over encrypted data, but its overhead in both computation and communication can be prohibitively high, and HE faces some practical issues with nonarithmetic computation. MPC allows different users to jointly compute a function over everyone's input without anyone learning others' inputs, but it scales poorly due to multiple rounds of communication. 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 at scale due to high paging overhead and limited memory space, especially at the scale demanded in the industry by large companies such as Facebook [4]. While these technologies can be applied for use in computation over aggregated user data, each one has disadvantages for the extreme scale and high complexity of modern data analytics, leading us to consider a hybrid approach.
Traditionally, the research areas of TEEs 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 TEEs and HE.

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

The popularly used Intel SGX TEE 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 [2], 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 [39], thus the performance impact and other downsides of total memory encryption technologies are not yet well-known. In summary, SGX supports smaller general secure computations efficiently but is limited in its performance at scale.

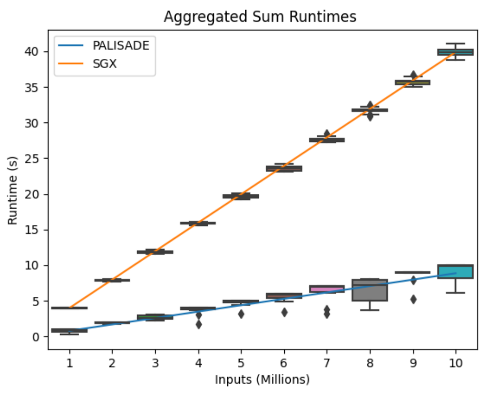


Fig. 1: Comparing Simple Sum Calculations. (PALISADE implementation used depth-1 CKKS with $N = 512$.)

HE cryptosystems allow computation over encrypted data. The most functional HE schemes are Fully Homomorphic Encryption (FHE) schemes, which allow arbitrary computation [31, 12, 18]. 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 [37], with a high potential for parallelization and optimizations [45, 8, 35]. 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 on encrypted data cannot be computed 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 [68, 74]. 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). 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.[3] 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 that HE is superior to SGX under certain circumstances, which indicates the need for using integrated solutions for scalable secure computation solutions.

[3] The source code of these experiments can be found at `https://github.com/justinpajak/Sum_SGX` and `https://gitlab.com/ColinMcKechney/homomorphic-sum`.

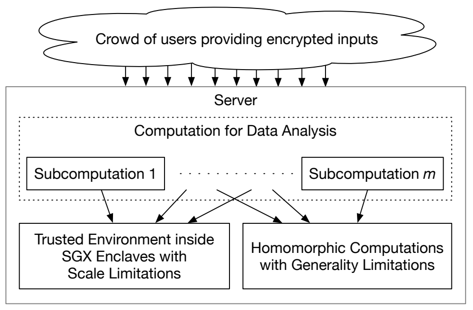

Crowd of users providing encrypted inputs
Server
Computation for Data Analysis
Subcomputation 1
Subcomputation *m*
Trusted Environment inside SGX Enclaves with Scale Limitations
Homomorphic Computations with Generality Limitations

Fig. 2: Overview of Our Integration

There exist some lines of research seeking to combine the capabilities of cloud-based HE and TEEs to cover each strategy's disadvantages [21, 74, 28, 32, 65, 60, 16]. Prior related work combining TEEs and cloud-based HE [60, 16] 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 [68], is used to integrate a HE library, **P**ALISADE [55], and a popular implementation of TEEs, Intel **S**GX [22]. 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 [60, 21], 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 [75, 50, 51, 26, 34, 9].
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 [68] 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.

Acknowledgements

This work was supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA) via contract #2020-20082700002. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor. The authors also thank Drs. Kurt Rohloff and Yuriy Polyakov (Duality Technologies) for their advice on PALISADE.

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) variants, which are limited by multiplicative depth.
The DGHV [70], TFHE [19], and FHEW [29] FHE schemes are FHE schemes that operate on single-bit plaintexts or batches thereof. However, having single-bit or bit-encoded plaintexts is highly cumbersome for many real-world applications.
The CKKS scheme [18] uses a fixed-point encoding to do approximate encrypted arithmetic, and is similar to the BGV [12] and B/FV [31] schemes. 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.
Using FHE has some pitfalls:

1. Ciphertext expansion: FHE ciphertexts may be much larger than plaintexts. VISE [21] showed ciphertext expansion factors of $2000\times$ for TFHE and up to $\approx 10^{12}\times$ for DGHV.
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 [74]. Smaller parameters can improve runtime at the cost of depth (discussed next), and batching can improve throughput, but the intensity is inherent in the schemes.
3. Depth: FHE schemes can only perform evaluation up to a certain multiplicative depth, at which point bootstrapping is needed to refresh the ciphertext. Using SHE variants avoids this overhead, but limits 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.

Parallel computing, cloud computing, and hardware acceleration can also be used to accelerate FHE. Various optimizations in lattice-based cryptography have greatly improved HE's efficiency and throughput at scale [35, 8, 17, 45]. Distributed computing can be applied to homomorphic computations, especially with special hardware such as GPUs [72, 25, 24], FPGAs [59, 69, 58, 43], and others [57, 66].

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. The memory of the process is held in a secure enclave that is encrypted when paged out, so even ring 0 processes cannot read it. A widely used realization of a TEE is Intel SGX [22].

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 [33].
2. Memory limit: Intel SGX has a physical memory limit of 128MB, due to the limited size of Processor Reserved Memory [33]. The practical limit is closer to 90MB [22]; using a greater amount of memory may incur untenable overhead from frequent paging. Some newer CPUs support a larger enclave size [47], 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: Many practical attacks have been demonstrated [75, 50, 51, 26, 34, 9], 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 [75], so to improve security it may be desirable to run SGX applications with only a single thread, though this may hurt performance.

SGX Efficiency According to our prior experiments, for the small-scale arithmetic computations that can fit into the capacity of SGX enclaves, SGX is 2-3 orders of magnitude faster than homomorphic encryption. The overhead of SGX increases up to 2000 times [6, 7, 44] at high scale because expensive paging and context switches occur [63]. It is nontrivial even for the manufacturer to increase enclave memory limits due to the integrity tree overhead and other constraints [63]. 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 and paging [60, 21].

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 [74] 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 [62] 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.
Because SGX may be vulnerable to side-channel attacks [13], simply keeping the secret key privately inside the memory enclave is not sufficient to guarantee security. 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. TEEFHE's experiments only used fewer than 30 clients.

3.2 VISE

The VISE system [21] is another effort to combine TEEs and HE for cloud-based secure computation. In HE, branching on encrypted conditions cannot be easily done, and large ciphertext expansions may make it infeasible for sensor nodes on slower networks (e.g., satellite Internet) to send 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 for computation. 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 was deployed with 870 sensor node clients.
VISE modifies the original source code of implementations of the DGHV [70] and TFHE [20] FHE schemes to make them SGX-friendly.

3.3 Other Similar Work

Other lines of work use an SGX for data management and trusted computing for use with other cryptographic primitives. The Iron system [32] 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. [65] 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. [71] and Luo et al. [46] apply SGX-based protocol management and secure computing for private auctions and private user matching, respectively. Karl et al. [40] use a TEE to provide noninteractivity for general-purpose secure multiparty computation, and later works of Karl's [42, 41] 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 [60, 16] use SGX and additive homomorphic encryption for secure queries on patient data, showing an increase over solely using SGX.

4 GPS and its Instantiation

4.1 Scenario and Assumptions

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 [54] 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.

We consider the threat model of an honest-but-curious adversary who may eavesdrop on 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 is released publicly, approaches such as differential privacy [30] should be used to protect user privacy. We thus 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 [64]: 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 $f(X)$.

Fig. 3: Ideal Functionality of GPS for Secure Computation.

Definition 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 [53, 73, 14, 52, 38]. 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 [27, 28].

4.2 Scheme and Technology Choices

C and C++ are the languages of choice for SGX programming, so we did not consider HE libraries in other languages such as Lattigo [49]. The most prominent C++ homomorphic encryption libraries are Microsoft SEAL [62], HElib [36], and PALISADE [55], all of which implement efficient polynomial ring arithmetic using Residue Number System [35, 8, 17] and Number-Theoretic Transform [45] techniques. Of those three, PALISADE implements the most schemes, provides a very wide array of additional functionality such as signatures and identity-based encryption [61], is extensively documented with examples, and is the most

actively developed and supported. 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 improved throughput, which is necessary for high-scale computations. While our implementation uses CKKS, the system can easily use BGV, B/FV, or TFHE, all of which are supported by PALISADE.
PALISADE and its dependencies cannot be used directly in SGX, as SGX applications can use only a limited set of the C and C++ standard libraries. We thus explore a general and reusable method of porting FHE libraries to SGX. The Graphene containerization framework [67, 68] 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. By using Graphene, we can easily run unmodified PALISADE applications in SGX, without any need to modify PALISADE. This combination of technology - Graphene, PALISADE, and SGX - comprises our system, GPS, that we can apply to computations not easily handled by SGX or HE alone.

4.3 Security (Informal)

GPS is secure according to Definition 1. 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 (disregarding 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.

4.4 Methodologies/Strategies of Integration

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 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 close to $O(1)$ but difficult to compute homomorphically (i.e., conditions, high multiplicative depth, or nonarithmetic computation) 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 computation can be described in terms of its abstract dataflow (e.g., as a directed acyclic graph), 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).

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.

4.5 Instantiation: Linear Regression Training

As an instantiation of our idea to be used in evaluation, we chose the training of linear regression models. We do not focus on the optimization of linear regression training; this case is an example to show concrete ideas and experimental 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. 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 Batching thousands of operands into a single ciphertext greatly improves HE's throughput. [10, 11]. 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. 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.) In our application, we can have all n users 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 placed 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. 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

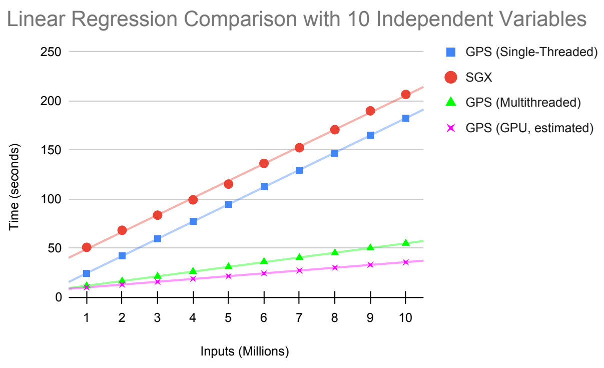
Our test workstation used an Intel Xeon CPU with 20 cores operating at 3.7GHz and 128GB memory. Our tests, written in C++, used the Development version of PALISADE, version 1.11.2. Our source code is available at `https://github.com/justinpajak/LinReg_SGX/` and `https://gitlab.com/palisade/graphene-palisade-sgx`.

5.2 Experimental Results

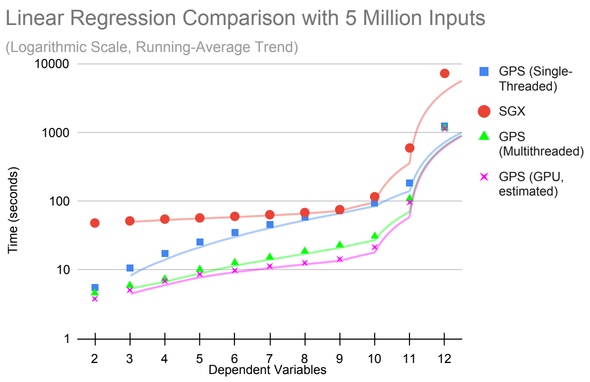
We evaluated scenarios with varying parameters, allowing p to vary from 2 to 12, and n to vary from $1,000,000$ to $10,000,000$, in increments of $1,000,000$. All results shown are the average of 10 trials, except at $n = 5,000,000$, $p = 12$, which used 5 trials due to high runtime. Standard deviations for results were within $\pm 2.2\%$ of the mean.
Setting $N = 8192$ 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. We thus only need two ciphertext moduli totalling 111 bits, giving this CKKS setting between 192 and 256 bits of (classical) security – greater than the 80 bits used in TEEFHE [74], and the 128 bits evaluated by VISE [21]. 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.

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 4a 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×. This comparison is for a large-scale multi-user computation; for smaller workloads SGX is likely to outperform GPS. Figure 4b 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 version of GPS without optimizations outperforms SGX, though both exhibit the same asymptotic behavior.

Multithreaded Performance Our evaluation in Section 5.2 used only a single thread for each program (though the first two matrix multiplications were computed in parallel). Using OpenMP [23], 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 $\mathbf{X}^T\mathbf{y}$. Figure 4 shows the performance achieved by applying PALISADE's multithreading to GPS, and Table 2 and Table 1 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



(a)



(b)

Fig. 4: Experimental Results for Linear Regression Performance. (a) Varying n (b) Varying p

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.

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 [5, 58, 25, 48, 57, 66, 72, 24, 59]. 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 [5]. 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 Figure 4, Table 2, and Table 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

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

for large-scale computations with data from a large number of users. 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

GPS 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. 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. Finally, GPS requires expert knowledge of both SGX and HE to implement correctly, which may discourage its use by non-experts.

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.

References

1. 37 Mind Blowing YouTube Facts, Figures and Statistics – 2021. `https://incomelords.com/youtube-statistics/`. Accessed: 2021-09-26.
2. 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`.
3. 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.
4. Facebook Research 2021 Privacy Enhancing Technologies request for proposals, Apr 2021.
5. A. Al Badawi et al. Privft: Private and fast text classification with homomorphic encryption. *IEEE Access*, 8:226544–226556, 2020.
6. S. Arnautov et al. {SCONE}: Secure linux containers with intel {SGX}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 689–703, 2016.
7. M. Bailleu et al. {SPEICHER}: Securing lsm-based key-value stores using shielded execution. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 173–190, 2019.
8. J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca. A full rns variant of fv like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 423–442. Springer, 2016.
9. A. Biondo et al. The guard's dilemma: Efficient code-reuse attacks against intel {SGX}. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1213–1227, 2018.
10. M. Blatt et al. Optimized homomorphic encryption solution for secure genome-wide association studies. *BMC Medical Genomics*, 13(7):1–13, 2020.
11. M. Blatt, A. Gusev, Y. Polyakov, and S. Goldwasser. Secure large-scale genome-wide association studies using homomorphic encryption. *Proceedings of the National Academy of Sciences*, 117(21):11608–11613, 2020.
12. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.
13. F. Brasser et al. Software grand exposure:{SGX} cache attacks are practical. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.
14. F. Brasser et al. Dr. sgx: Automated and adjustable side-channel protection for sgx using data location randomization. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 788–800, 2019.
15. L. Cheng, F. Liu, and D. Yao. Enterprise data breach: causes, challenges, prevention, and future directions. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(5):e1211, 2017.
16. W. Chenghong et al. Scotch: Secure counting of encrypted genomic data using a hybrid approach. In *AMIA Annual Symposium Proceedings*, volume 2017, page 1744. American Medical Informatics Association, 2017.
17. J. H. Cheon et al. A full RNS variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*, pages 347–368. Springer, 2018.
18. J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.

19. I. Chillotti et al. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *international conference on the theory and application of cryptology and information security*, pages 3–33. Springer, 2016.
20. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/.
21. L. Coppolino, S. D'Antonio, V. Formicola, G. Mazzeo, and L. Romano. Vise: Combining intel sgx and homomorphic encryption for cloud industrial control systems. *IEEE Transactions on Computers*, 70(5):711–724, 2021.
22. V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.
23. L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.
24. W. Dai, Y. Doröz, and B. Sunar. Accelerating ntru based homomorphic encryption using gpus. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2014.
25. W. Dai and B. Sunar. cuHE: A homomorphic encryption accelerator library. In *International Conference on Cryptography and Information Security in the Balkans*, pages 169–186. Springer, 2015.
26. F. Dall, G. De Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom. Cachequote: Efficiently recovering long-term secrets of sgx epid via cache attacks. 2018.
27. N. Drucker and S. Gueron. 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*, pages 85–88, 2017.
28. N. Drucker and S. Gueron. Achieving trustworthy homomorphic encryption by combining it with a trusted execution environment. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, 9(1):86–99, 2018.
29. L. Ducas and D. Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 617–640. Springer, 2015.
30. C. Dwork. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation*, pages 1–19. Springer, 2008.
31. J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012.
32. B. Fisch et al. Iron: functional encryption using intel sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 765–782, 2017.
33. A. T. Gjerdrum, R. Pettersen, H. D. Johansen, and D. Johansen. Performance of Trusted Computing in Cloud Infrastructures with Intel SGX. In *CLOSER*, pages 668–675, 2017.
34. J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.
35. S. Halevi, Y. Polyakov, and V. Shoup. An improved rns variant of the bfv homomorphic encryption scheme. In *Cryptographers' Track at the RSA Conference*, pages 83–105. Springer, 2019.
36. S. Halevi and V. Shoup. Helib. *Retrieved from HELib: https://github. com. shaih/HElib*, 2014.
37. K. Han, S. Hong, J. H. Cheon, and D. Park. Logistic regression on homomorphic encrypted data at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9466–9471, 2019.

38. Z. Hongwei et al. Tsgx: Defeating sgx side channel attack with support of tpm. In *2021 Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS)*, pages 192–196. IEEE, 2021.
39. Intel Corporation. *Intel Architecture Memory Encryption Technologies*, 4 2021. Rev. 1.3.
40. R. Karl, T. Burchfield, J. Takeshita, and T. Jung. Non-interactive MPC with trusted hardware secure against residual function attacks. In *International Conference on Security and Privacy in Communication Systems*, pages 425–439. Springer, 2019.
41. R. Karl et al. Cryptonomial: A framework for private time-series polynomial calculations. In *Securecomm*, 2021. `https://eprint.iacr.org/2021/473`.
42. R. Karl, J. Takeshita, and T. Jung. Cryptonite: A framework for flexible time-series secure aggregation with online fault tolerance. In *Securecomm*, 2021. `https://eprint.iacr.org/2020/1561`.
43. S. Kim et al. Hardware architecture of a number theoretic transform for a bootstrappable rns-based homomorphic encryption scheme. In *IEEE FCCM 2020*, pages 56–64. IEEE, 2020.
44. R. Kunkel, D. L. Quoc, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. Tensorscone: a secure tensorflow framework using intel sgx. *arXiv preprint arXiv:1902.04413*, 2019.
45. P. Longa and M. Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *International Conference on Cryptology and Network Security*, pages 124–139. Springer, 2016.
46. J. Luo, X. Yang, and X. Yi. Sgx-based users matching with privacy protection. In *Proceedings of the Australasian Computer Science Week Multiconference*, pages 1–9, 2020.
47. D. Martin. Intel xeon ice lake cpus to get sgx with expanded security features, Oct 2020.
48. T. Morshed et al. CPU and GPU accelerated fully homomorphic encryption. In *IEEE HOST*, pages 142–153. IEEE, 2020.
49. C. Mouchet et al. Lattigo: A multiparty homomorphic encryption library in Go, 2020.
50. K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482. IEEE, 2020.
51. A. Nilsson, P. N. Bideh, and J. Brorsson. A survey of published attacks on intel sgx. *arXiv preprint arXiv:2006.13598*, 2020.
52. H. Oh et al. 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*, pages 1903–1918, 2020.
53. O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 227–240, 2018.
54. C. Peikert and S. Shiehian. Multi-key fhe from lwe, revisited. In *Theory of Cryptography Conference*, pages 217–238. Springer, 2016.
55. Y. Polyakov, K. Rohloff, and G. W. Ryan. PALISADE lattice cryptography library user manual. *Cybersecurity Research Center, New Jersey Institute ofTechnology (NJIT), Tech. Rep*, 15, 2017.
56. M. Prater. 25 Google Search Statistics to Bookmark ASAP. `https://blog.hubspot.com/marketing/google-search-statistics`. Accessed: 2021-09-26.

57. D. Reis et al. Computing-in-memory for performance and energy-efficient homomorphic encryption. *IEEE TVLSI*, 28(11):2300–2313, 2020.
58. M. S. Riazi, K. Laine, B. Pelton, and W. Dai. HEAX: An architecture for computing on encrypted data. In *ACM ASPLOS*, pages 1295–1309, 2020.
59. S. S. Roy et al. Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *2019 IEEE HPCA*, pages 387–398. IEEE, 2019.
60. M. N. Sadat et al. Safety: secure gwas in federated environment through a hybrid solution. *IEEE/ACM trans. on computational biology and bioinformatics*, 16(1):93–102, 2018.
61. A. Sahai and B. Waters. Fuzzy identity-based encryption. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 457–473. Springer, 2005.
62. Microsoft SEAL (release 3.0). `http://sealcrypto.org`, Oct. 2018. Microsoft Research, Redmond, WA.
63. M. Taassori, A. Shafiee, and R. Balasubramonian. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *ACM ASPLOS*, pages 665–678, 2018.
64. J. Takeshita, R. Karl, and T. Jung. Secure single-server nearly-identical image deduplication. In *IoTSPT-ML at ICCCN 2020*. IEEE, 2020.
65. J. Takeshita, R. Karl, A. Mohammed, A. Striegel, and T. Jung. Provably secure contact tracing with conditional private set intersection. In *Securecomm*, 2021.
66. J. Takeshita, D. Reis, T. Gong, M. Niemier, X. S. Hu, and T. Jung. Algorithmic acceleration of b/fv-like somewhat homomorphic encryption for compute-enabled ram. In *International Conference on Selected Areas in Cryptography*. Springer, 2020.
67. C.-C. Tsai et al. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.
68. C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library {OS} for unmodified applications on {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 645–658, 2017.
69. F. Turan, S. S. Roy, and I. Verbauwhede. Heaws: An accelerator for homomorphic encryption on the amazon aws fpga. *IEEE Transactions on Computers*, 69(8):1185–1196, 2020.
70. M. Van Dijk et al. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 24–43. Springer, 2010.
71. J. Wang, N. Lu, Q. Cheng, L. Zhou, and W. Shi. A secure spectrum auction scheme without the trusted party based on the smart contract. *Digital Communications and Networks*, 2020.
72. W. Wang et al. Accelerating fully homomorphic encryption using gpu. In *2012 IEEE conference on high performance extreme computing*, pages 1–5. IEEE, 2012.
73. W. Wang et al. 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*, pages 2421–2434, 2017.
74. W. Wang et al. Toward scalable fully homomorphic encryption through light trusted computing assistance. *arXiv preprint arXiv:1905.07766*, 2019.
75. N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In *European Symposium on Research in Computer Security*, pages 440–457. Springer, 2016.