FLARE: A Fast, Secure, and Memory-Efficient Distributed Analytics Framework

Xiang Li
Tsinghua University
lixiang20@mails.tsinghua.edu.cn

Fabing Li
Xi'an Jiaotong University
lifabing@stu.xjtu.edu.cn

Mingyu Gao
Tsinghua University
Shanghai Artificial Intelligence Lab
Shanghai Qi Zhi Institute
gaomy@tsinghua.edu.cn

ABSTRACT

As big data processing in the cloud becomes prevalent today, data privacy on such public platforms raises critical concerns. Hardware-based trusted execution environments (TEEs) provide promising and practical platforms for low-cost privacy-preserving data processing. However, using TEEs to enhance the security of data analytics frameworks like Apache Spark involves challenging issues when separating various framework components into trusted and untrusted domains, demanding meticulous considerations for programmability, performance, and security.

Based on Intel SGX, we build FLARE, a fast, secure, and memory-efficient data analytics framework with a familiar user programming interface and useful functionalities similar to Apache Spark. FLARE ensures confidentiality and integrity by keeping sensitive data and computations encrypted and authenticated. It also supports oblivious processing to protect against access pattern side channels. The main innovations of FLARE include a novel abstraction paradigm of shadow operators and shadow tasks to minimize trusted components and reduce domain switch overheads, memory-efficient data processing with proper granularities for different operators, and adaptive parallelization based on memory allocation intensity for better scalability. FLARE outperforms the state-of-the-art secure framework by 3.0× to 176.1×, and is also 2.8× to 28.3× faster than a monolithic libOS-based integration approach.

PVLDB Reference Format:
Xiang Li, Fabing Li, and Mingyu Gao. FLARE: A Fast, Secure, and Memory-Efficient Distributed Analytics Framework. PVLDB, 16(6): 1439 - 1452, 2023.
doi:10.14778/3583140.3583158

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tsinghua-ideal/flare.

1 INTRODUCTION

The prominent paradigm of cloud computing allows cost-efficient and convenient computational outsourcing to much more powerful remote servers. However, sending computations to the cloud inevitably exposes user data to the public, raising severe security concerns especially with security-sensitive applications that process private data [14]. Widely used data analytics frameworks, e.g., Apache Spark [84], enable high-performance distributed processing, but do not provide native data privacy protection. Modern cryptographic algorithms, such as homomorphic encryption [20] and secure multi-party computation [24, 83], could be applied on top of these frameworks, but would incur unacceptably huge overheads with over 1000× slowdown [15, 46, 54]. Alternatively, commercially available hardware-based trusted execution environments (TEEs), such as Intel SGX [32] and ARM TrustZone [1], provide us a more practical solution. They rely on hardware *enclaves* with attestation and isolation, resulting in much smaller overheads than cryptography-based approaches.

However, building a fast and secure data analytics framework with Intel SGX is not trivial. A monolithic method to directly integrate Spark into enclaves [3, 6, 75] is undesired. The substantially enlarged trusted computing base (TCB) would include libOS and JVM, and expose many vulnerabilities [35, 74]. Some essential features of the distributed framework would be discarded due to limited support [70, 75] and/or high performance overheads. We therefore follow a minimalist philosophy which *carefully separates and coordinates the framework components across the trusted and untrusted domains*. Doing so requires us to rethink the framework design paradigm and come up with new optimizations, in order to keep similar *programmability and functionalities*, alleviate various *performance* overheads, and ensure strong *security* guarantees.

We propose a fast, secure, and memory-efficient distributed data analytics framework, FLARE, which leverages Intel SGX to offer confidential and verifiable outsourcing computations against malicious adversaries. FLARE offers a familiar user programming interface similar to Apache Spark. With the key principle of *trusted/untrusted domain separation*, FLARE keeps the Spark RDD abstraction as well as various complicated management and scheduling modules in the untrusted domain. This minimizes the TCB, and also retains most desired functionalities such as *distributed scheduling, fault tolerance, and data persistence/caching*. In the trusted domain, FLARE uses a novel abstraction paradigm of *shadow operators* to process data inside enclaves. A shadow operator can be efficiently associated to and shared by different RDDs of the same type outside enclaves.

FLARE incorporates novel performance optimizations to alleviate various SGX overheads, such as cross-domain context switches, data footprints exceeding trusted cache/memory regions, and contention incurred by intensive memory allocation. Shadow operators support *serialization-free data transfers* across domains by maintaining consistent memory layouts and known data types. They could also be *fused into shadow tasks* to further reduce the overheads of

Proceedings of the VLDB Endowment, Vol. 16, No. 6 ISSN 2150-8097.
doi:10.14778/3583140.3583158

domain switches and data transfers. FLARE transparently manages data processing in a *memory-efficient* manner. It applies different data processing granularities for different execution phases, such as *block-level streaming* for local computation and *sub-partitioning* for global shuffling. FLARE also *adaptively adjusts its degrees of multi-thread parallelization* based on the intensity of memory allocation, to avoid contention and improve scalability.

For security, FLARE supports both an *encryption mode* for data content protection, and a stronger *oblivious mode* to further address the access-pattern-related side-channel vulnerabilities in SGX at the cost of higher performance overheads. Specifically, while the aforementioned performance optimizations are all applicable to the encryption mode, the oblivious mode cannot use the optimized sub-partitioning shuffling and must fall back to the more expensive oblivious sort primitive. In addition, to ensure dynamic execution integrity even in the presence of an untrusted task scheduler, FLARE uses novel *cooperative integrity guards* across the distributed enclaves, logging and verifying their runtime traces.

FLARE is implemented in Rust, a high-performance and memory-safe language, widely recommended for secure systems. We evaluate FLARE on both the newest and the old SGX processors with different trusted memory capacities, across a wide range of data analytics, graph processing, and machine learning benchmarks. Compared with two state-of-the-art secure data analytics frameworks, Opaque [88] and SGX-PySpark [40], FLARE achieves 3.0× to 176.1×, and 13.4× to 469.4× speedups, respectively. The security overhead, e.g., slowdown over the corresponding insecure counterpart, is reduced by 5.4× in FLARE over Opaque. FLARE is also 2.8× to 28.3× faster than a monolithic design that directly uses libOS porting [69]. Even when compared to several optimized insecure baselines, FLARE only incurs up to 8.8× slowdown, and sometimes even outperforms Apache Spark due to language advantages. We also evaluate and analyze FLARE scalability under both multi-core and multi-node settings.

2 BACKGROUND AND MOTIVATIONS

FLARE is a fast and secure data analytics framework that relies on hardware-based trusted execution environments (TEEs). We first introduce the background on distributed analytics frameworks and TEEs. Then we highlight the challenges of combining the two into an efficient design, under the threat model assumed in this work.

2.1 Distributed Analytics Framework: Spark

Spark is a widely used *distributed* data analytics framework [84]. It supports various interfaces built on top of Spark Core, including Spark SQL, Spark Streaming, MLlib (for machine learning), and GraphX (for graph processing), making it convenient to write parallel programs on multi-core machines and multi-node clusters. Spark leverages in-memory computation, thus exhibiting superior performance compared to traditional MapReduce engines [10].

The fundamental data abstraction in Spark is *resilient distributed datasets* (RDDs), which can be transparently partitioned across multiple compute nodes. RDDs are immutable. When programming, the user creates a series of RDDs, each representing a parallel operation such as `map`, `groupBy`, or `reduceByKey`. Some of them may embed

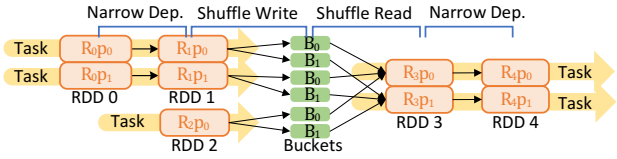


Figure 1: Narrow and shuffle dependencies in Spark.

function literals (closures). For example, `map(_.split(" "))` transforms an RDD to a new one by splitting each string by spaces. Such a series of RDD transformations form a *lineage* that reflects the dependencies among them.

There are two types of dependencies capturing the relationship of partitions in adjacent RDDs, as shown in Figure 1. The *narrow* dependency indicates that at most one partition in a child RDD is derived from each parent RDD partition, so that the partitions can be processed independently, such as in parallel or by streaming. The *shuffle* (a.k.a., wide) dependency is more complex, where more than one child RDD partitions depend on a parent RDD partition. Therefore the processing of shuffle dependency is typically divided into two steps, shuffle write and shuffle read. During shuffle write, a partitioner acts on each parent RDD partition to generate multiple buckets. The number of buckets and their data distribution are determined by the child RDD partitions. For example, in Figure 1, the child RDD3 has two partitions, so the partitioner generates two buckets B_0 and B_1 per each partition of RDD1 and RDD2 by hashing their keys. Then, in shuffle read, the child RDD fetches from these buckets to compose its partitions, e.g., the R_3p_0 partition fetches from all the B_0 buckets.

Spark applies *lazy evaluation*, i.e., RDDs are not computed and materialized until an *action* is invoked to produce some externally visible results (e.g., `count` the number of elements or `save` to the disk). The leader node in the cluster constructs a *job*, described by a directed acyclic graph (DAG) of relevant RDD lineage. The job DAG is further divided at shuffle dependencies into *stages*. At the beginning of each stage, the leader determines the number of partitions, and then spawns and dispatches one *task* per partition to the compute nodes, as shown in Figure 1.

Spark supports *persistence* by storing RDDs either to external disks or in memory (i.e., *caching*). Persisting/caching an RDD can exclude the previous transformations from the task, avoiding repeated computation and materialization. Spark also provides *fault tolerance*. If a partition is lost, it can be recovered from a persisted copy (checkpoint) if one exists, or is simply recomputed according to the lineage information.

2.2 Trusted Execution Environment: SGX

Intel Software Guard eXtension (SGX) is an instruction set architecture (ISA) extension to support hardware-based TEEs [2, 29, 50]. SGX provides a set of instructions on a trusted processor to securely create, manage, and destroy a trusted *enclave* within an untrusted computing platform. An enclave is created by loading the authentic user program and verifying its content through *attestation*. Afterwards, even privileged software, including the operating system (OS) and the virtual machine manager, can never steal/distort the

Table 1: Normalized performance overheads of sequential and random data accesses in the trusted and untrusted domains, respectively. Measured on Scalable SGX.

Domain	Untrusted		Trusted	
Access Pattern	Sequential	Random	Sequential	Random
Perf. Penalty	1×	up to 10×	≈ 1×	up to 15×

sensitive data or tamper with the code. The user then supplies encrypted data to the program through an `ECALL`, which enters the enclave and invokes the trusted code, decrypting and processing the data. An `OCALL` can be performed within the `ECALL` to temporarily leave the enclave and use system services (e.g., system calls) from the untrusted OS. These context switches across domains (a.k.a., *domain switches*) incur large performance overheads (Section 2.3).

At runtime, whenever enclave data need to leave the processor package, e.g., evicted from on-chip caches to off-chip memory, SGX hardware transparently performs data encryption and authentication [28, 31, 34], increasing the memory access latency [3]. Legacy versions of SGX used costly Merkle tree structures to protect off-chip memory content freshness. The newest SGX on recent Scalable Intel processors (called "Scalable SGX" in our paper) has eliminated this overhead, but still exhibits up to 15× performance penalty for cache misses from the trusted domain, significantly higher than that of the untrusted accesses, as shown in Table 1. In addition, SGX uses a processor-reserved trusted memory region called *enclave page cache* (EPC) to store enclave code and data. Scalable SGX supports a large EPC size such as 1 TB [34], but the legacy SGX only allows a small EPC space, e.g., 128 MB [33], which causes huge *secure paging* overheads if the application memory footprint exceeds the EPC limit [3, 79]. Unfortunately, many practically deployed systems still support only the legacy SGX.

Dynamic memory allocation is now supported by SGX [49]. The current SGX SDK only ports Doug Lea's dlmalloc and Google's tcmalloc into enclaves [32]. Compared to the untrusted world, designing an efficient memory allocator for enclaves is non-trivial and needs additional optimizations due to various requirements. For security, SGX does not allow multiple enclaves to share memory space, so `mmap` support must be restricted [68]. Synchronization and other security bugs must be avoided in the allocator implementation [80]. For performance, more expensive cache/EPC misses make cache locality optimizations more critical [27], and `OCALL` invocations in the allocator incur extra domain switch cost [79].

Despite the theoretically secure specifications, existing hardware implementations of SGX still suffer from various *side-channel* vulnerabilities, including access-pattern-based attacks [26, 41, 64, 77, 82] and power/electromagnetic signal analysis [19, 47, 87]. Therefore, additional programming efforts and/or software-level protections must be incorporated [12, 71, 88]. Cryptographic primitives to hide access patterns, such as oblivious RAM (ORAM) [23, 25, 58, 73], can be integrated into TEEs [9, 45, 62], but would incur unacceptable performance overheads up to several orders of magnitude. Some enclave architectures provide hardware oblivious memory (OM) [18, 71], where the access patterns to data within the OM cannot be observed externally. OM typically makes use of the on-chip cache space, with extra locking and cleansing to ensure clean states before use and remain non-interference until released. OM allows for more efficient oblivious execution than generic ORAM [9, 45]. But different from ORAM which can protect the whole memory space, the OM size is typically limited, e.g., only a few MBs.

2.3 Motivations and Challenges

It is possible to use existing approaches like libOS to directly integrate Spark into SGX with minor modification [3, 6, 69, 75]. However, such a solution is actually neither secure nor efficient. Spark has a large code base with various complex components including the heavy Java virtual machine (JVM) runtime. Naively porting everything into enclaves would substantially enlarge the trusted computing base (TCB) with higher vulnerability [35, 74]. It also stresses the limited trusted memory space (cache, OM, EPC, etc.) with large memory footprints, causing significant performance degradation. Note that instruction cache misses are typically worse than data cache misses, because processor front-end stalls are much more difficult to hide with microarchitectural optimizations than back-end stalls. Furthermore, ensuring obliviousness for both code and data accesses also becomes more difficult using the limited hardware OM, and may inevitably require the even more expensive ORAM primitive.

We therefore adopt the minimalist philosophy, i.e., *place the minimally necessary components into the enclave, and coordinate the framework components across the trusted and untrusted domains*. While promising, such a design needs careful considerations for *programmability*, *performance*, and *security*.

The **programmability and functionalities** of the secure framework should ideally be similar to the original insecure Spark for ease of use. Users would like to write similar high-level (e.g., functional) programs as before. The *interaction between trusted and untrusted components* should be handled automatically and efficiently by the framework. The framework should also support various features such as *distributed execution, fault tolerance, and data persistence and caching*, which are essential for practical large-scale data analytics. Integrating them with SGX is non-trivial as the data and their processing are now spread across multiple distributed nodes each containing both trusted and untrusted domains.

SGX incurs several major **performance** overheads that must be alleviated in the framework design. First, the separation of framework components would lead to frequent interaction between the trusted and untrusted domains, with expensive *domain switches* [13, 79, 86]. The accompanying *data transfers*, including necessary encryption/decryption and potentially also (de)serialization, further exacerbate the latency. Second, data analytics applications typically process large datasets of GBs to TBs. The legacy SGX with a limited EPC size would suffer from excessive *secure paging* [3, 79]. Although Scalable SGX [34] with a large EPC is free from secure paging, it still exhibits large overheads from *processor cache misses* that introduce extra encryption/authentication.

Third, in this work we also observe a specific performance bottleneck of *memory allocation* when porting Spark-like computation into SGX. Our target framework is allocation-intensive. The immutability of RDDs implies that new output objects must be created for each RDD transformation, which could involve heavy memory allocation. However, current allocator implementations in SGX

Table 2: Multi-thread scalability of SGX on two micro-benchmarks with different amounts of memory allocation. The input is a vector of a billion 32-bit integers.

`map(|x|vec![x])`		`map(|x|x+1)`	
1 thread	8 threads	1 thread	8 threads
31.8 s	688.8 s	0.61 s	0.09 s

Table 3: Correlation between allocation intensity and speedup of 8 threads over 1 thread, when using SGX to execute representative tasks in our benchmarks.

Alloc. Intensity	0.004	0.01	0.02	1.1	2.4	3.2
Speedup	8	4.6	1.9	0.19	0.11	0.087

severely limit the multi-thread scalability. Table 2 shows the execution time of two micro-benchmarks running on our servers that support Scalable SGX; `map(|x|vec![x])` (Rust syntax) needs to allocate a vector for every data item, while `map(|x|x+1)` only allocates once for the entire array. It is clear that excessive memory allocation would destroy multi-thread scalability due to the heavy contention on the shared user-space heap and the allocator internal states [32, 86], even resulting in over 20× *slowdown* at 8 threads. We further run a diverse set of tasks from our benchmarks (Section 7.2). We correlate the speedup of 8 vs. 1 thread with the *allocation intensity*, defined as the memory allocation count per input data item. The results in Table 3 show higher allocation intensity leads to lower speedup and even serious slowdown.

Finally, the **security** of the full system needs to be carefully enforced. By moving some complex but critical framework components, e.g., the task scheduler, out of the trusted domain, it is possible that the *execution integrity* could be compromised if the generated job DAG has been tampered with. It is insufficient to simply rely on the confidentiality and integrity protection of each individual enclave in the distributed system; tasks with wrong control flow and/or wrong input data could be dispatched from the malicious scheduler. In addition, due to the significance of side-channel vulnerabilities, *oblivious processing* becomes a common requirement for SGX development [16, 52, 62, 88]. It is desired that our framework could also support such a higher level of security.

2.4 Threat Model and Security Guarantees

The user locally owns a trusted client machine. At the server side which consists of a distributed cluster with multiple compute nodes, only the SGX-capable hardware processors on these nodes are trusted. We assume a powerful malicious attacker, who is able to control all privileged software on each server including the OS and the virtual machine manager, and wants to steal sensitive information or mislead the client through incorrect or stale results. The attacker can arbitrarily observe and tamper with the code and data in the untrusted devices, including memories, disks, and networking. However, any curious or malicious adversarial behaviors to the trusted enclaves will fail because of the guarantees of SGX.

We consider two modes with different security levels.

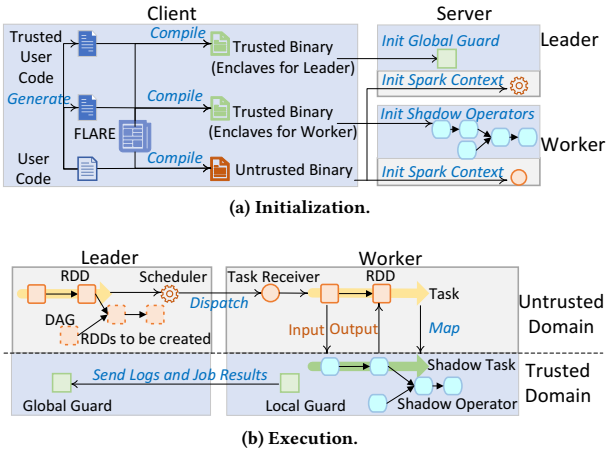


Figure 2: System overview of Flare.

The **encryption mode** protects computation confidentiality and integrity. It ensures that the privacy of user data values is never compromised, and the result received by the client must either be correct or explicitly indicate the execution has been attacked. Although the encryption mode does not consider side-channel issues, it is still a practical and commonly applicable scenario, since it offers a reasonable balance between performance and security.

The **oblivious mode** additionally considers access-pattern-based side channels on top of the encryption mode, where the attacker has the ability to infer sensitive information from the access traces, including memory and disk access addresses on each single machine, and network traffic among distributed nodes. This mode guarantees that these access traces are irrelevant to the sensitive data content. More formally, let I be the input data (of size S_I and with element type T_I), P be the program (e.g., a physical plan in SQL), O be the output result (of size S_O and type T_O). The public parameters are $\texttt{params} = (S_I, S_O, T_I, T_O, P)$ which are irrelevant to the sensitive data values. The access traces on the specific input data I are defined as $\texttt{Trace}(\texttt{params}, I)$. The oblivious mode ensures that, for all I, O, and P, there exists a polynomial-time simulator `Sim` such that $\texttt{Sim}(\texttt{params}) \simeq \texttt{Trace}(\texttt{params}, I)$, i.e., a computationally indistinguishable trace can be simulated without knowing the data content. Supporting for obliviousness enables stronger security, but usually at the cost of significant performance overheads.

Other timing-based attacks [7, 30] (e.g., externally observable execution time), as well as denial-of-service (e.g., powering off the server cluster), are considered as out of scope.

3 DESIGN OVERVIEW

We build a fast, secure, and memory-efficient data analytics framework, Flare, to address the challenges in Section 2.3. Figure 2 illustrates the system overview and the overall workflow of the Flare framework. Flare is written in Rust, and users still use Spark-like APIs such as `map` and `groupBy` to write programs. Flare supports both the *encryption mode* and the *oblivious mode* defined in Section 2.4, with different performance vs. security tradeoffs. More specifically, the encryption mode in Flare uses a number of

performance optimizations to address the inefficiencies of domain switches, data transfers, and memory allocation, while only a subset of these techniques are safely applicable to the oblivious mode. We will clearly specify the conditions as we introduce each technique.

Following the *minimalist philosophy*, Flare leaves most framework logic in the untrusted domain, including the complicated task scheduler and the immutable RDD abstraction; it only ports the functions that directly process sensitive data into enclaves. Note that the separation is *not* about data. All data in Flare are treated as sensitive and must be processed in the enclaves unless otherwise specified. All sensitive data, when kept by RDDs outside the enclaves, are encrypted and authenticated.

Such a framework separation has several benefits. First, it allows Flare to maximally preserve the desired rich functionalities in the original framework. With the same RDD abstraction and execution flow in the untrusted domain, distributed execution, fault tolerance, and data persistence/caching can be easily supported without intertwining with the SGX mechanisms. Second, the TCB of Flare is also reduced. The scheduling and management code is typically complicated and vulnerability-prone [8, 36, 61], so excluding them from the trusted domain could greatly improve security, as well as performance (reducing instruction cache miss cost in the trusted domain, Section 2.2). The trusted binary, including the framework code, has a small size around 4 MB, well within the cache/OM limit, benefiting performance and simplifying oblivious execution.

Flare leverages a novel paradigm called *shadow operators* to efficiently process plaintext data inside the trusted domain (Section 4). Conceptually, they are a bunch of atomic operators initialized inside the enclave and invoked dynamically by the corresponding RDDs from outside the enclave at runtime. Shadow operators effectively reduce SGX domain switch overheads, by enabling *serialization-free* cross-domain data transfers and by being fused into *shadow tasks*.

Flare also supports *memory-efficient data processing* (Section 5). For various execution phases such as narrow computations and shuffle operations, Flare uses different *data granularities* to ensure the currently processing data are within the cache/OM limit. Flare also adjusts the *degrees of parallelization*, i.e., the number of threads used on each node, to avoid contention in memory allocation.

Flare carefully ensures security (Section 6). It uses *cooperative integrity guards* across enclaves to prevent the untrusted framework scheduler from compromising execution integrity. Its oblivious mode avoids *access pattern leakage*, and also alleviates specific *control-flow side channels* [60, 81].

We next summarize the common workflow of initialization and execution for *both encryption and oblivious modes* in Flare.

Initialization. As shown in Figure 2a, the user first prepares the program by compiling her code with the Flare framework at the local trusted client machine, which produces trusted and untrusted code to run inside and outside enclaves, respectively. This process uses standard compiler tools plus a simple, custom source-to-source translation script in Flare (Section 4.1). Then she submits the compiled binaries to the server cluster. All compute nodes correctly load the trusted part into their enclaves, ensured by remote attestation [2, 4, 63].

In the untrusted domain, the leader and the workers initialize a set of basic structures similar to those in Spark, e.g., runtime contexts, a scheduler, and a cache tracker. In the trusted enclaves, the workers construct the shadow operators in correspondence to the RDDs outside, as well as the local integrity guards. The leader additionally creates a global integrity guard in its enclave. These components will be used in the ways described below.

Execution. As in Figure 2b, the execution flow in the untrusted domain stays almost the same as Spark (Section 2.1). RDDs are lazily evaluated. An action triggers the leader to build a job DAG of several stages. At each stage, the scheduler generates and distributes tasks to the workers. Upon receiving a task, the worker executes it using the shadow operators in its enclave. It copies the encrypted RDD data into the enclaves, decrypts them, executes a shadow task on the data, encrypts the results, and transfers them out to materialize the RDDs in the untrusted domain. At the same time, the local guards inside the enclave record the task execution metadata. The detailed flow is elaborated in Section 4. Once the worker completes all dispatched tasks, it holds the results, waiting for the next stage to fetch from it, or sends back to the leader if it is the final stage in the job. Moreover, it securely sends the task execution metadata to the global guard in the leader enclave. The global guard checks the information against the desired execution flow it holds.

4 SHADOW OPERATORS & SHADOW TASKS

Prior secure computing systems (e.g., Opaque [88]) have also adopted the philosophy of trusted and untrusted domain separation. A straightforward way is to prepare a set of generic, *abstract* operators (e.g., `map`) inside the enclave, and *individually* invoke them when their corresponding RDD transformations are applied on the sensitive data [88]. This approach has several inefficiencies. First, it incurs *frequent domain switches and data transfers* when invoking the abstract operator for *every* RDD. Second, abstract operators cannot perceive the concrete data types and function closures. For example, both `map(_.split(" "))` and `map(i=>i+1)` in user programs invoke the same in-enclave operator `map`, despite the different data types and closures. Therefore, abstract operators require *interpreted execution and data serialization*, i.e., taking a sequence of bytes as input, deserializing the data, and interpreting the closure to execute, which would incur large overheads. Third, although abstract operators may allow for more generic programming of the user enclave code, it still requires the user to specify and interpret any user-defined functions (UDFs) [39].

In Flare, we propose a novel abstraction called *shadow operators* for the trusted domain to resolve the above inefficiencies. The key idea is to generate *customized in-enclave operators that embed the concrete information such as data types and function closures*, rather than using generic and abstract operators, thus supporting serialization-free cross-domain data transfers and interpretation-free execution (Section 4.1). Instead of individually executing each shadow operator, we can dynamically *fuse* shadow operators into *shadow tasks* according to the tasks assigned from the scheduler, thus eliminating excessive data copy and encryption/decryption across the enclave boundary (Section 4.2). The techniques in this section apply to *both the encryption and oblivious modes*.

4.1 Shadow Operators

Formally, shadow operators are *mutable function templates* in the trusted domain, and are the counterpart to the immutable RDDs

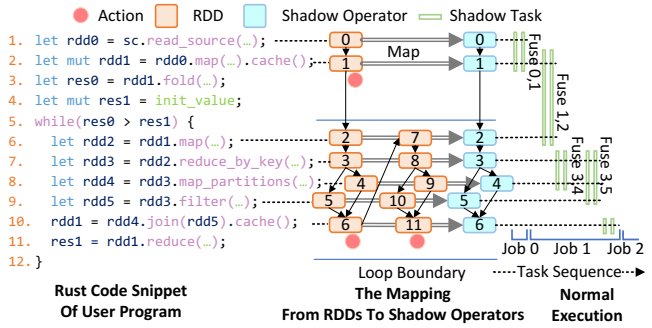


Figure 3: FLARE code execution example. RDDs are mapped to shadow operators, which are then fused to shadow tasks.

in the untrusted domain. Each shadow operator is denoted by $(ID, T_{\text{in}}, T_{\text{out}}, Op, F, \sigma)$, where T_{in} and T_{out} are the input/output data types, Op and F stand for the abstract operator (e.g., `map`) and the closures (i.e., UDF, if any). σ denotes a set of parameters that are the only mutable part in the template. It includes the dependencies, the number of partitions of its corresponding RDD, and any variables captured by the closures. While RDDs are dynamically created at runtime, shadow operators are *built statically and deterministically at initialization*. For example, `rdd2` in Figure 3 may not be created if the `while` loop never executes, but shadow operator 2 is always constructed inside the enclaves. As a result, certain parameters in the shadow operator need to be mutable (see below).

To construct shadow operators, FLARE provides a simple source-to-source script to process the user program (e.g., remove all control flow statements like `while` and `if` and keep all basic blocks). Therefore all shadow operators could be built in all worker enclaves at the beginning. The processed program is compiled together with the FLARE framework to generate the trusted code running in enclaves (Figure 2a). Other than the script, the rest of the compilation uses standard compilers (e.g., `rustc`) without extra modification. The original user program, in particular its control flow, is still used to create RDDs in the untrusted domain.

Such a static build flow of shadow operators in FLARE greatly simplifies the execution management. If shadow operators were built on demand during execution, extra synchronization and bookkeeping would be necessary to track which operators have been prepared in each worker. Ensuring execution integrity would become difficult in such a dynamic setting.

Statically building shadow operators also implies that a shadow operator could be *shared* by multiple RDDs of the same type. Figure 3 shows an example of the mapping from RDDs to shadow operators. In the untrusted context, separate RDDs are created for multiple iterations of the same loop at runtime. In contrast, inside the trusted enclaves, only five shadow operators are statically constructed, each of which is shared by the RDDs at the same code location but created in different loop iterations. Furthermore, shadow operator sharing also saves enclave space.

Sharing requires shadow operators to be mutable, while RDDs in FLARE are still immutable for fault tolerance reasons. When invoked, a shadow operator is associated with a specific RDD outside. The output data, after encrypted, are sent out to materialize this RDD. Again, ID, T_{in}, T_{out}, Op, F are fixed after compilation, while others (e.g., dependencies) could change according to the associated RDD. For example, shadow operator 2 may depend on RDD 1 or 6 (both referring to the variable `rdd1` in the code) in Figure 3.

Finally, owing to the embedded information, shadow operators enable *serialization-free data transfers* across the two domains and *interpretation-free execution* in the enclave. In previous designs such as Opaque [88], data (de)serialization is necessary when transferring data across the enclave boundary. This is because the data types cannot be known beforehand and the two domains may use different memory layouts. For example, in Opaque, the untrusted part is written in Scala, while the trusted part is in C++, and they use `flatbuffer` to encode data type information. In contrast, FLARE keeps the information about data types and closures in the shadow operators, and the code components in both domains are written in Rust so they use the same memory layout.

4.2 Fusing into Shadow Tasks

To avoid separately invoking individual shadow operators from the untrusted domain with excessive domain switches, FLARE *fuses* the shadow operators that execute together into a *shadow task* within one `ECALL`. Data could thus be directly passed from one shadow operator to the next one inside the enclave in plaintext, without repetitive cross-domain transfers and encryption/decryption.

Though fusion is a standard optimization, previous frameworks (e.g., Opaque [88]) cannot do it efficiently because type systems cannot be shared across isolated domains, resulting in interpretation and serialization. To support fusion, FLARE combines static compilation and dynamic invocation. At compilation, shadow operators are specially constructed to embed types and closures (Section 4.1). The actual fusion happens dynamically at execution time. When a worker receives a task dispatched from the scheduler, it passes the sequence of the involved shadow operator IDs to the enclave through a single `ECALL`. Each shadow operator in the shadow task does not immediately materialize its output; instead, it lazily hands over the iterator to the next, in order to improve data locality (Section 5.1). More specifically, the enclave code starts from the shadow operator indexed by the last ID. When the last shadow operator needs its input from the previous one, the previous shadow operator is invoked, and in turn asks for its previous one. The shadow operator indexed by the first ID is finally encountered, and it creates an iterator on the input data. Then the direction turns backward. The intermediate shadow operators transform the iterator, and the final one would consume the iterator and generate the output.

In Spark, users may specify a `cache` point to any intermediate RDD. FLARE supports this by materializing the data in the middle of the shadow task and storing to the (encrypted) RDD in the untrusted domain, so other workers could fetch it through network. The enclave code fully consumes the iterator when encountering a `cache` point, and creates a new iterator on the materialized data for further transformations.

FLARE automatically performs rule-based fusion. First, fusion does not cross task boundaries, since the shuffle between two tasks involves interaction among multiple enclaves. As in the example of Figure 3, shadow operator 2 could only be fused with 1 (and 0, if 1 is not cached), due to the shuffle before 3 (`reduceByKey`). Second,

if adjacent operators process data with different granularities (Section 5), they are also not fused. Some operators like `mapPartitions` need to process the whole partition for correct functionalities, while other operators in the same task can do element-wise execution.

5 MEMORY-EFFICIENT DATA PROCESSING

As discussed in Section 2.3, even with the new Scalable SGX, processor cache misses (and EPC paging in the legacy SGX) cause severe performance degradation, which advocates in-cache computation. The oblivious mode also requires data to fit in the limited hardware OM. Moreover, memory allocation greatly impacts the scalability of in-enclave threads.

To solve the above issues, FLARE achieves *memory-efficient data processing* by carefully adjusting *data granularities* and *parallelization degrees* when executing different phases. Currently, in Spark and FLARE, the job DAG is divided into stages at the points of shuffle dependencies, and each stage contains multiple tasks, one per data partition (Section 2.1). As a result, most tasks in FLARE consist of three phases, *shuffle read* (if any), *narrow computation*, and *shuffle write* (if any), as shown in Figure 1. When tasks are executed in enclaves, there are extra *decryption* and *encryption* phases. All these phases exhibit different memory access and allocation characteristics, where FLARE handles them using specific techniques. Specifically, to restrict memory footprints within the caches/OM, we apply two data granularities, *block-level streaming* and *sub-partitioning*, to narrow and shuffle operations, respectively (Sections 5.1 and 5.2). To alleviate the memory allocation scalability issue, we propose *allocation-adaptive parallelization*, by only selectively enabling multi-thread processing for beneficial phases (Section 5.3), without designing a new memory allocator. All these managements in FLARE are transparent to user programs.

This section first discusses all the optimizations applicable to the encryption mode. For the oblivious mode, some techniques need to be adjusted, specifically the shuffle operation in Section 5.2, which will be discussed in details in Section 6.2.

5.1 Block-Level Streaming for Narrow Ops

Data in FLARE are encrypted and authenticated in the unit of *blocks*, whose size balances between encryption/authentication overheads (preferring larger block sizes) and fine-grained control capabilities (preferring smaller block sizes). The latter is desired because shuffle operations need to extract individual data items (Section 5.2). We allow user programs to configure the block size, and also empirically find a default 64 kB (typically thousands of items) works well in most cases. Each data partition in FLARE contains many such blocks.

For the narrow computation phase of a task, the computations on different data items are highly independent. This is the nature of narrow dependencies, i.e., each output partition only depends on one input partition. Therefore, we can simply process each block in a *streaming* manner, minimizing the memory footprint in the enclave. All blocks are handled in one `ECALL` with fused shadow operators, sequentially pulling each block into the enclave for processing.

5.2 Sub-Partitioning for Shuffle Ops

Unfortunately, streaming is not suitable for shuffle operations due to the all-to-all data dependencies. When the partition size exceeds the

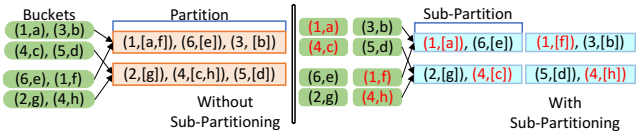


Figure 4: Sub-partitions affect `groupBy`. Because keys 1 and 4 are split into inconsistent sub-partitions in the two partitions, they cannot be properly grouped.

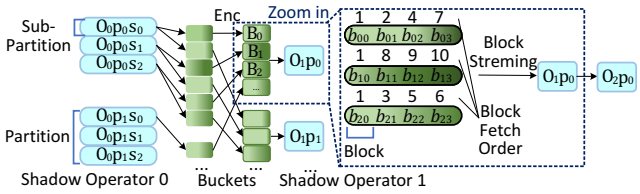


Figure 5: Liebig's law fetch. The gradually changing green colors from light to dark indicate the ascending key order.

cache/OM capacity limit, a straightforward approach is to further divide them into smaller *sub-partitions* [67]. A sub-partition consists of many blocks. Each time we keep one sub-partition in the enclave and generate intermediate buckets from it during shuffle write, similar to Figure 1 but now sub-partitions substitute partitions.

However, a naive sub-partitioning implementation would affect shuffle correctness as illustrated in the example of Figure 4. This is because the sub-partitions are coarsely divided from an unsorted partition so the data keys are arbitrarily distributed among them, unlike the partitions that are grouped by the keys. If we separately shuffle each sub-partition, the same keys in different sub-partitions would not be correctly grouped into one output partition.

Liebig's law fetch. To ensure correct shuffle, we use a technique named *Liebig's law fetch* that applies extra sorting on data keys. As in Figure 5 left, when generating buckets from each sub-partition during shuffle write, we sort each plaintext bucket inside the enclave (illustrated using gradually changing colors), before encrypting and sending them out. Note that the sub-partition is sized to fit in the cache/OM (discussed later), and so does the sorting of the buckets.

During shuffle read, we collect all relevant buckets generated from different input sub-partitions, and fetch one block from the beginning of each bucket into the enclave (Figure 5 right). Because each bucket is sorted, these blocks contain the smallest keys among all buckets. So we can aggregate the items in the ascending key order, until they fill up a block, and are further processed with block-level streaming. The rest items remain in the enclave. When we exhaust one input block, we fetch the next from that bucket. This follows the Liebig's law, i.e., always fetching from the shortest remaining bucket. The aggregation on a certain key is finished and the newly formed block can be further processed only when the smallest remaining keys in all buckets are larger than it, thus solving the issue in Figure 4. Note that at the beginning, we must fetch all the first blocks from all buckets, though there may be some bucket that has larger keys (e.g., b_{10} of the 2nd bucket) than the later blocks in another bucket (e.g., b_{01} of the 1st bucket). Only after decryption, can we know the actual order among them.

Sub-partition sizing. Liebig's law fetch requires us to keep at least one block from each bucket in the enclave. However, the number of buckets could be large after applying sub-partitioning, because instead of per partition, now we generate buckets per sub-partition. This requires us to use larger sub-partition sizes while still staying within the cache/OM limit. We observe that shuffle operations usually exhibit robust and predictable memory usage proportional to the input data size, because they do not produce new data but simply reorganize existing data. This allows for a simple approach to assemble blocks into a sub-partition in the enclave after the narrow computation, until the size of the sub-partition exceeds a pre-defined threshold (e.g., less than the cache size).

5.3 Allocation-Adaptive Parallelization

Memory allocation remains as a critical scalability bottleneck in Flare, as illustrated in Section 2.3. Note that using mutable shadow operators in Flare instead of immutable RDDs does not decrease the allocation intensity. The generated results of each shadow operator invocation must be separately allocated as usual. Shadow task fusion and block-level streaming also do not affect the allocation count; they only alter the order of allocating different data items.

The intuitive way to overcome limited allocation scalability is to design or port a new allocator. However, there are many more challenges beyond performance involved in a comprehensive allocator design as discussed in Section 2.2. Actually, even in the untrusted domain, there are many allocator designs such as ptmalloc [22], tcmalloc [21], and jemalloc [17], with different tradeoffs in various aspects. People make different choices in different scenarios.

Therefore, in Flare we opt for a higher-level approach that can work for any specific allocator implementation. Flare *adapts to different workloads based on their allocation intensity*, i.e., only enabling multi-thread parallelization for those with light memory allocation, while restricting parallelization when heavy memory allocation occurs. Such allocation-adaptive parallelization is compatible to different allocators, including dlmalloc and tcmalloc in the current SGX SDK as well as those that might be ported in the future.

More specifically, recall that a (shadow) task contains multiple phases including decryption, shuffle read, narrow computation (fused shadow operators), shuffle write, and encryption. We can classify each phase as either heavy or light in terms of allocation intensity by using a pre-determined threshold (a framework hyperparameter). All phases could be either heavy or light, except that the encryption phase is light since the outputs are always byte arrays. We then allow (resp. forbid) multi-thread parallelization for the light (resp. heavy) phases. We find that for the current dlmalloc and tcmalloc in SGX, it is best to always use the maximum number of threads for light phases and a single thread for heavy phases. For more complex allocators, it is possible to set multiple thresholds for finer increments of parallelization degrees.

Two issues remain: how to set the threshold, and how to measure the allocation intensity of a phase. Given that different allocators exhibit diverse scalability, we use per-allocator thresholds determined from the empirical correlation similar to Table 3, i.e., setting the threshold to the point between speedup and slowdown. We use online profiling to measure allocation intensity. Flare counts the allocation for each phase of a dispatched task on the first few

Figure 6: Shuffle write decomposition and pipelining. D, R, N, W, E stand for decryption, shuffle read, narrow computation, shuffle write, and encryption, respectively. Shuffle write is decomposed into sorting (S) and aggregation (A).

block/sub-partition samples to determine the heavy vs. light classification. Then it applies the configuration on the rest data.

Optimizations for shuffle. We further optimize shuffle operations in Flare. With our Liebig's law fetch optimization in Section 5.2, shuffle operations in Flare heavily rely on sorting to generate buckets during the shuffle write phase. We observe that sorting can be made in-place without memory allocation, thus eliminating the bottleneck. Specifically, we decompose the shuffle write phase into a sorting phase plus an aggregation phase. Each sub-partition is first in-place sorted in the enclave before generating the buckets. As a result, we know directly how many items should go to each bucket, and can pre-allocate the bucket space once. After that, we copy the items into the buckets, and (optionally) simultaneously aggregate the same keys (now continuously stored) into partial results to reduce the communication cost in the later shuffle read phase. The shuffle read phase is kept the same as Section 5.2.

The decomposition has two benefits. First, it reduces memory allocation during bucket generation. Second, as shown in Figure 6, the zero-allocation sorting phase can now be made overlapped (i.e., pipelined) with the other heavy phases before and after it (narrow computation, aggregation). These heavy phases only use a single thread. As sorting does not introduce contention, it is free to leverage the otherwise idle cores in the system.

6 SECURITY ENHANCEMENTS

6.1 Cooperative Integrity Guards

In Flare, shadow operators are dynamically fused into shadow tasks according to the tasks dispatched from the untrusted scheduler. Such a design exposes an execution integrity vulnerability, where the adversary who controls the scheduler could execute a task composed of any operators as well as a sequence of arbitrary tasks. Note that it is completely legal for a shadow operator to be fused with different other ones in different tasks; in Figure 3, shadow operators 0 and 1 are fused in job 0 triggered by `fold`, while 1 (already cached) and 2 are fused in job 1 for `reduce`.

To ensure the execution indeed follows the authentic control flow, we propose *cooperative integrity guards* in all enclaves. At runtime, the worker and leader enclaves cooperatively use their local and global guards, to verify the dynamic execution flow. More specifically, we include a copy of the user program as part of the leader enclave so it gets attested during initialization (Figure 2a). The global guard in the leader enclave then analyzes the program and builds the same job DAG as the scheduler would do. Specifically, each task in the DAG is represented by $(IDs, \sigma s, in, out)$. IDs denotes the sequence of shadow operators executed in order in this task. Recall for each shadow operator, all parameters are

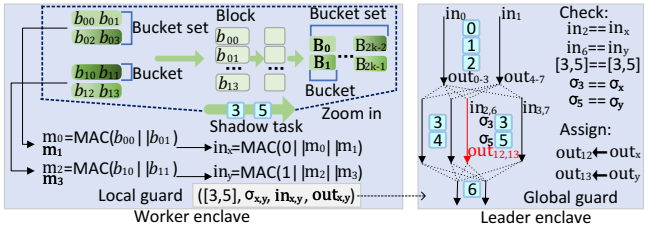


Figure 7: Cooperative integrity guards in leader and worker enclaves. The DAG corresponds to the 2nd job in Figure 3.

fixed at compilation and attested during initialization, except for σ. Therefore, we only need to verify σ at runtime to ensure the integrity of each individual shadow operator. *in* and *out* denote the input/output data MACs, respectively. The user supplies *in* of the input at the beginning, e.g., in_0 and in_1 in Figure 7.

The global guard waits for the local guards in the worker enclaves to submit execution logs. In the worker enclave, the local guard records the metadata of each executed shadow task ($IDs, \sigma s, in, out$) as explained above, and securely sends the log to the global guard, where it is checked against the corresponding DAG, as shown in Figure 7. The checks include (1) whether each σ is valid (for the integrity of each shadow operator); (2) whether IDs and σs compose a valid fused shadow task (for the integrity of the shadow task); (3) whether the input data match the output data of the predecessor tasks, i.e., no data blocks are dropped, and their MACs match (for the integrity between tasks). Notice that because SGX enclaves run attested code, as long as the input data MAC is correct and the shadow operator fusion is valid, the integrity of the output data MAC is guaranteed. If the verification passes, the global guard regards this task as successfully executed, and unlocks the successor tasks depending on it by properly propagating the data MACs. The procedure continues until the last stage, where the MAC of the final result is sent to the user as an authentication.

In Flare, both domains will generate the DAG, but the DAGs contain different information. In the untrusted domain, the DAG contains details for various execution management, e.g., scheduling and fault tolerance. In the trusted domain, the generated DAG is thinner, mainly for integrity checks. The user program included in the leader enclave is usually much smaller than the framework code, so the TCB would be still small.

Recall that Flare computes MACs in the unit of data blocks (Section 5.1) instead of partitions. This fine granularity may result in many MAC messages and increase the communication cost between the workers and the leader. We propose hierarchical authentication in Flare, as shown in Figure 7. Specifically, in Flare, when buckets are generated from sub-partitions and encrypted in the unit of blocks at the end of a task, we compute the MAC of a bucket by hashing the concatenation of the block MACs belonging to this bucket. For example, in Figure 7, a task in the previous stage computes the bucket MAC m_0, which serves as an input MAC of the current task. Similarly, since a partition contains multiple sub-partitions, we compute the bucket set MAC by hashing the concatenation of the bucket MACs, e.g., in_x in Figure 7. The bucket set MAC is the final MAC sent to the global guard in the leader. When the current task fetches the relevant buckets, the local guard first verifies the block MACs (i.e., b_{00}) and decrypts the data blocks, then computes the bucket MACs and the bucket set MACs. MAC computations run in parallel with data block computations.

Different from Opaque [88] and VC3 [63], the cooperative guards are designed to work with the memory optimizations in Flare. Moreover, Opaque [88] tracked execution and input/output data at each *operator* granularity, while Flare does so per *task*. This is because no intermediate results between internal shadow operators leave enclaves, so we only need to record the operator IDs and the input/output MACs of the entire task. Moreover, the lightweight check procedure is overlapped with the computation in the workers, and thus the cost is mostly hidden. Per-task MAC generation incurs negligible (<1%) cost measured in our experiments.

6.2 Oblivious Mode

Hiding access patterns. Flare's basic architecture (e.g., shadow operators and integrity guards) supports both the encryption and oblivious modes. However, the shuffle-related optimizations (Section 5.2) rely on value-based partitioning, which does not satisfy the stronger security requirements in Section 2.4. For example, the communication volume between nodes depends on the private data content. Flare uses similar oblivious algorithms as in Opaque [88] in its oblivious mode. We first briefly illustrate the algorithms in Opaque, which are *not* our contribution. We then discuss how to adapt the encryption mode techniques to support obliviousness.

The core building block in Opaque is *oblivious sort* across all the workers, replacing shuffle write and read. Operations involving shuffle are transformed into oblivious sort and scan [88]. For example, for aggregate, oblivious sort is first performed to explicitly group items with the same key, and then each worker conducts a single scan to aggregate locally. Since a group may cross workers, inter-node communication is needed to derive the final aggregated values. We leave algorithm-level optimizations to future work.

In Flare, Liebig's law fetch (Section 5.2) is only secure for the encryption mode, but the relevant sub-partitioning can be leveraged to optimize local oblivious sort. Given the existence of hardware oblivious memory (OM), Flare first performs quicksort on each sub-partition fitting in the OM. Then it applies bitonic sort [53] to merge the sorted sub-partitions. Moreover, in the oblivious mode where sorting dominates performance (>80% in our evaluation), we statically decide to only parallelize the sorting and encryption phases, while the other phases use single-thread.

Mitigating control-flow side channels. For single-job execution, the control flow is deterministic. However, for general Spark applications, there may exist statements such as `if` and `while` with predicates involving secret variables, triggering different jobs and causing distinguishable access patterns. To mitigate this issue, we use *dummy execution*. Specifically, it is the global guard that is responsible for resolving secret predicates. For `if` branches, the untrusted scheduler is told to execute both branches, but only the correct result is selected by the global guard. For loops, the global guard asks the untrusted scheduler to launch more iterations until reaching a constant threshold r. Such execution padding hides sensitive loop iterations. Users may flexibly specify r in Flare to balance security levels and performance overheads.

7 EVALUATION

7.1 Implementation

FLARE is implemented in Rust (about 25k LOC) on top of Vega [65], an open-source re-implementation of Apache Spark in Rust. Rust offers advantageous language features like memory safety [44], zero-cost abstraction, and functional programming. While Vega has only limited support for all the features in Spark, it is able to provide competitive performance. We use the Rust SGX SDK v1.1.3 [11, 78] for the framework code in the trusted domain. FLARE uses the AES-GCM scheme to encrypt/decrypt when data leave/enter enclaves. The relevant keys in different enclaves and the secure communication channels are established during enclave attestation.

7.2 Experimental Setup

Platform. To evaluate FLARE, we use three types of machines: (1) Local **Scalable SGX**, an Intel Xeon Gold 5317 processor of 3 GHz and 64 GB EPC. (2) Local **legacy SGX**, an Intel Core i7-9700K processor of 3.6 GHz and 128 MB EPC. Both these two local machines run Ubuntu 18.04 with Linux kernel 5.4.0. (3) **Cloud instances** `ecs.g7t.4xlarge` on Alibaba Cloud, with Scalable SGX and inter-machine network bandwidth of up to 25 Gbps. We use the local Scalable SGX as the default platform for detailed evaluation.

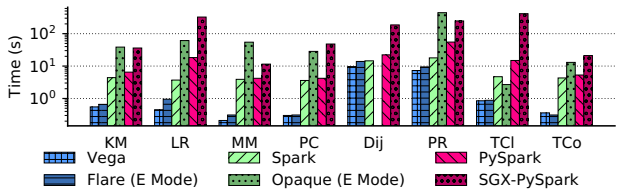
Workloads. We use a diverse set of benchmarks from data analytics, graph processing, and machine learning, including PageRank (`PR`), Transitive Closure (`TCl`), Dijkstra (`Dij`), Triangle Counting (`TCo`), K-Means (`KM`), Logistic Regression (`LR`), Matrix Multiplication (`MM`), and Pearson Correlation (`PC`). They are widely used in other frameworks [12, 63, 67, 88], and cover both sequential and random access patterns. We use real-world datasets [42, 43, 48], with large memory footprints up to several tens of GB on a single node. We also use TPC-H at scale factor 1 as a SQL benchmark. We choose scale 1 because larger datasets are too slow to run on the Opaque baseline due to the lack of memory-efficient optimizations. Our results show FLARE exhibits higher benefits on larger datasets.

Baselines. Besides Spark [84] and Vega [65], we use four other Spark-like systems as baselines. PySpark is an interface for Apache Spark in Python. Vega-Occlum directly puts Vega into Occlum v0.27.1 [69], an SGX libOS. Opaque [88] implements operators of Spark with C/C++ in SGX. SGX-PySpark [40] combines SGX and PySpark using SCONE [3]. Vega-Occlum, Opaque, and SGX-PySpark provide security, respectively based on Vega, Spark, and PySpark. We evaluate the encryption mode by default unless otherwise stated, as Opaque does not open source the oblivious mode.

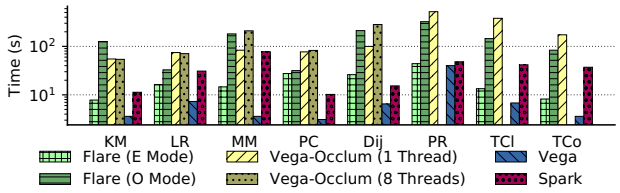
7.3 Single-Machine Overall Performance

Figure 8 shows the overall performance comparison between FLARE and the baselines. Besides the default large datasets of tens of GBs, we also use smaller datasets (10 to 50 MB) because some baselines cannot run the large datasets (e.g., JVM errors) or have extremely slow speeds. The small datasets are still larger than the on-chip cache capacity. All experiments run on one Scalable SGX machine with a sufficient EPC size even for the large datasets. `Dij` is not implemented in Opaque because its UDFs are not supported.

In Figure 8a with the small datasets, we use single-thread. Among the three secure frameworks, FLARE largely outperforms Opaque by



(a) Small datasets (10 to 50 MB); single-thread.



* The missing bars have run for at least an hour and finally crash.

(b) Large datasets (5 to 30 GB); best multi-thread.

Figure 8: Overall performance on a single machine.

3.0× to 176.1×, and is also 13.4× to 469.4× faster than the monolithic integration in SGX-PySpark. Some of these advantages are from their different insecure counterparts (Vega for FLARE, Spark for Opaque, PySpark for SGX-PySpark). PySpark is consistently slower than Spark; Vega, despite not fully optimized, runs faster than Spark due to the language advantages. Nevertheless, even if we only look at the slowdown within each pair of secure/insecure frameworks, the security overhead of FLARE (i.e., $t(\text{FLARE})/t(\text{Vega})$) is still 5.4× smaller on average than that of Opaque, not yet considering Opaque re-implemented Spark in faster C/C++. FLARE is even slightly faster than Vega on `TCo` due to our extra memory-efficient optimizations.

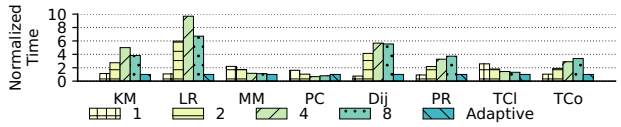
We now analyze the significant improvements of FLARE over Opaque. First, Opaque executes each operator separately, incurring repetitive data transfer overheads including encryption/decryption and (de)serialization. FLARE instead uses shadow operator fusion and serialization-free transfers (Section 4). Second, FLARE integrates various memory-efficient techniques (Section 5), which reduce cache misses and off-chip secure data accesses. Opaque also inefficiently triggers unnecessary actions to ensure timely RDD materialization before processing inside enclaves. In contrast, SGX-PySpark fuses all operators in one Python process inside the enclave, which avoids frequent domain switches. Therefore, despite being based on the slower PySpark framework, SGX-PySpark sometimes outperforms Opaque, but is still consistently slower than FLARE.

The experiments on the large datasets in Figure 8b are allowed to use multi-thread execution up to 8 threads. We only use Vega and Spark with their best 8-thread configurations as the insecure references; PySpark is always slower to Spark. For the secure frameworks, we find Opaque does not scale to large datasets and crashes for most experiments. For the few successful runs, Opaque is more than two orders of magnitude slower than FLARE. So we use another baseline, Vega-Occlum. We also evaluate the FLARE oblivious mode.

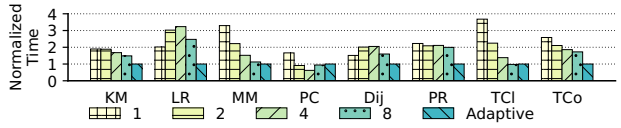
FLARE in its encryption mode achieves 2.8× to 28.3× speedups over the best multi-thread configurations (1 or 8 threads) of Vega-Occlum, which directly ports Vega with the Occlum libOS [69].

Table 4: Flare slowdown on legacy SGX over Scalable SGX.

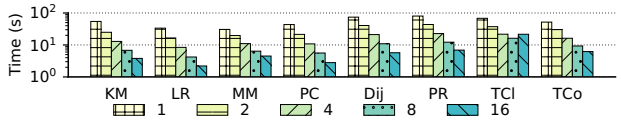
Mode	KM	LR	MM	PC	Dij	PR	TCl	TCo
E	0.9	0.7	1.3	0.6	0.9	1.5	1.2	1.6
O	3.5	0.7	12.3	0.6	4.8	3.7	31.9	14.9



(a) Multi-thread with dlmalloc, normalized to `Adaptive`.



(b) Multi-thread with tcmalloc, normalized to `Adaptive`.



(c) Multi-node distributed execution, in log scale.

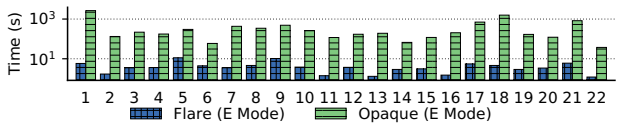
Figure 9: Multi-thread and multi-node scalability of Flare.

The minimalist philosophy in Flare results in better performance than the monolithic integration. Using large datasets and multi-thread execution also makes the memory-efficient optimizations and the allocation-adaptive parallelization in Section 5 more critical. Compared to the insecure multi-thread baselines, Flare only exhibits moderate slowdown of 1.1× to 8.8× over Vega, and 0.2× (even faster) to 2.7× over Spark. Some benchmarks, e.g., PC, show high security overheads in Flare. This is because they involve little computation, so the relative cost of data transfers (encryption, etc.) across enclave boundaries dominates performance. Finally, Flare oblivious mode is on average 6.5× slower than its encryption mode.

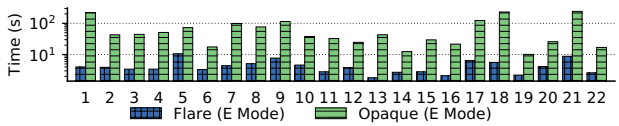
Compatibility on legacy SGX. Table 4 runs Flare on the legacy SGX machine and summarizes the slowdown compared to the default Scalable SGX machine. Generally the encryption mode performance is similar in both systems, demonstrating that Flare memory optimizations effectively overcome the EPC limit in legacy SGX. Some computation-intensive workloads (e.g., PC) actually run faster because of the higher CPU frequency. The slowdown mainly results from repetitive encryption/decryption to keep the data size within the EPC. The oblivious mode sees larger slowdown due to the extra rounds of sorting on the whole partitions. LR and PC do not involve shuffle and thus behave the same in both modes.

7.4 Multi-Thread/Multi-Node Scalability

We evaluate the scalability of Flare under both the multi-thread execution on a single machine, and the multi-node distributed setting. In Figures 9a and 9b, when using fixed 1, 2, 4, 8 threads without



(a) 8 threads on a single node.



(b) A cluster of 3 worker nodes, similar setting as [38].

Figure 10: Overall performance on TPC-H at scale factor 1.

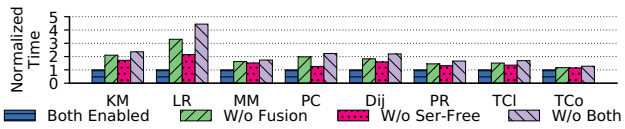


Figure 11: Benefits of shadow operators and shadow tasks.

adaption, Flare does not scale well, and may even slow down for some workloads (e.g., LR). Switching from dlmalloc to tcmalloc alleviates the issue [32] but still has poor scalability. Our adaptive parallelization in Section 5.3 selectively enables multi-thread execution for each phase, and reduces allocation contention during shuffle. It almost always achieves the best performance across all fixed degrees of parallelization for each workload. The implementation incurs at most 30% overheads, e.g., in Dij with dlmalloc.

We use cloud instances on Alibaba Cloud to evaluate the Flare scalability to multiple distributed nodes. As shown in Figure 9c, across different workloads, using 16 worker nodes achieves 3.2× to 15.2× speedups than using 1 node. Six out of the eight workloads have speedups larger than 8×. TCl has the worst scalability due to its heavy communication. For the Flare encryption mode, the communication cost among nodes is nearly the same as Spark except for small extra data integrity checksums; only each individual node runs slower due to interaction with the local enclave. Therefore the achieved scalability is reasonable.

7.5 TPC-H Performance

Given that Opaque exposes the higher-level SQL interface while Flare directly supports the low-level Spark Core, for a fair comparison, we additionally run the 22 queries of TPC-H. We keep the physical plans in both systems as similar as possible. The results are shown in Figure 10. In the single-node setting, the speedup ranges from 13× to 458×. In the multi-node setting, the speedup ranges from 4.6× to 55×. The main reasons for such performance advantages are similar to those explained in Section 7.3.

7.6 Benefits of Individual Techniques

Using shadow operators reduces data transfer overheads in Flare mostly in two ways: fusion decreases the number of necessary transfers, and serialization-free alleviates per-transfer cost. Figure 11

Table 5: Comparison of execution time (in seconds) between Flare and Vega-Occlum, demonstrating the effectiveness of cache-friendly designs.

Systems	KM	LR	MM	PC	Dij	PR	TCl	TCo
Flare (E mode)	13.9	32.9	30.5	54.3	36.2	75.1	30.5	12.4
Vega-Occlum	54.9	74.6	83.4	77	99.8	519.3	379.3	173.2
Speedup	3.9	2.3	2.7	1.4	2.8	6.9	12.4	13.9

quantifies these benefits. We disable either serialization-free or operator fusion, and both. Both techniques have significant impacts. When combined, they result in up to 4.4× and on average 2.1× better performance.

Then we evaluate the cache efficiency of Flare. Recall that Flare reduces costly instruction cache misses from the trusted domain by moving complex code out of enclaves, and reduces data cache miss overheads with memory-efficient data processing. We compare Flare with Vega-Occlum, which also uses Rust. As shown in Table 5, Flare cache-friendly designs bring 1.4× to 13.9× speedups. We argue that these benefits mainly come from their cache usage differences, not the other effects, becaues (1) the libOS integration in Vega-Occlum potentially reduces the ECALLs/OCALLs, and operator fusion is naturally done in Vega, from which point Flare is no better, and (2) here both systems use single-thread and the same memory allocator. For PC, no shuffle is involved, and the access pattern is sequential, so the 1.4× speedup mostly comes from the instruction cache miss reduction. For TCl and TCo, they exhibit more random access patterns, so the speedup of memory-efficient processing approaches 15×, matching the profiling in Section 2.2.

We also evaluate Liebig's law fetch. Without it, we uniformly fetch blocks from all buckets (e.g., b_{11} will be in the third fetch instead of b_{21} in Figure 5), and keep data inside the enclave until fully aggregated. This results in larger memory footprints. Liebig's law fetch contributes 1.3×, 1.8×, 1.6×, 3.1×, 1.8×, 3.8× speedups to KM, Dij, MM, PR, TCl, TCo, respectively. Other benchmarks do not involve shuffle. The speedup mostly depends on the data distribution. More precisely, it depends on the distribution of intermediate results after narrow computation. Without sorting in Liebig's law fetch, during shuffle read we will fetch unsorted arrays. That is, the data items with the same key are everywhere, so when aggregating them, there would be many cache misses. We observe that cache misses mostly occur in the hashmap/btreemap structures, which are used for aggregation.

8 RELATED WORK

There is a large amount of prior work integrating Intel SGX with data analytics frameworks for security. Platforms like SCONE [3], Graphene [75], SGX-LKL [56], and Occlum [69] allowed unmodified applications to run in enclaves, as convenient ways to build SGX-based secure data analytic systems. For example, SGX-Spark [59, 76] ported JVM into enclaves using SGX-LKL. SGX-PySpark [40] used SCONE to port partial JVM and also built a CPython/PyPy interpreter. However, both of them had a large TCB and might suffer from the security and performance issues discussed in Section 2.3. Others have built Spark or Hadoop-like secure frameworks without porting JVM. VC3 [63] built an in-enclave MapReduce engine, and Opaque [88] re-implemented Spark operators, both using C/C++. Without JVM, the TCBs of these two frameworks were small, and both of them assumed powerful threat models that ensure confidentiality and integrity against malicious attackers. VC3 provided region self-integrity invariants to avoid unexpected information leakage due to unsafe memory reads and writes. Opaque proposed oblivious relational operators to prevent access pattern leakage. However, none of the above work has comprehensively considered the performance issues associated with the limited cache/EPC capacities, the domain switch cost due to encryption and serialization, and the poor scalability. Flare leverages a better abstraction paradigm and various memory-efficient performance optimizations.

Apart from Spark or Hadoop, other proposals brought security into relational databases [5, 16, 52, 57], key-value stores [37, 72], and other workloads (e.g., SGX-BigMatrix for matrix computations [67]). Some of them also guaranteed obliviousness [16, 52]. SGX-BigMatrix had a blocking mechanism similar to sub-partitioning in Flare. ShieldStore [37] added a customized memory allocator inside enclave to support allocating untrusted memory without repetitive OCALLs.

In hybrid cloud scenarios, there is an opportunity to partition data into sensitive parts running on private clouds and non-sensitive parts on public clouds. Following this direction, Sedic [85] and SEMROD [55] built MapReduce-like frameworks, and [51] focused on joint query processing on the two types of data. Flare splits the code framework rather than the data, and thus is orthogonal.

Other TEEs like AMD-SEV [66] allow us to put Spark directly into enclaves, similar to the SGX libOS integration. This approach simplifies framework development with no need to separate components. Thus shadow operators and fusion are no longer necessary. However, it still suffers from larger TCB, larger memory footprints, lack of execution integrity, no obliviousness, and (likely) allocation scalability issues, which are resolved by FLARE. Besides, to pursue ultimate performance, we believe enclave models like SGX are still a good choice free of unnecessary code encryption.

9 CONCLUSIONS

We developed Flare as a distributed data analytics framework supporting confidential and verifiable computing with Intel SGX. Flare carefully separates the functionalities between the trusted and untrusted domains. In order to achieve both rich functionalities and high performance, Flare uses a novel abstraction paradigm called shadow operators, as well as various memory-efficient optimizations for data footprints and allocation scalability. With these improvements, Flare achieves 3.0× to 176.1× speedups over the state-of-the-art secure baseline, and only incurs up to 8.8× slowdown over the insecure system even on large datasets.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. We also thank Hongliang Tian for his helpful comments. This work was supported by the National Natural Science Foundation of China (62072262) and the Institute for Interdisciplinary Information Core Technology, Xi'an. Mingyu Gao is the corresponding author.

REFERENCES

[1] Tiago Alves. 2004. TrustZone: Integrated Hardware and Software Security. *White paper* (2004).

[2] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*.

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 689–703.

[4] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia. 2021. Avocado: A Secure In-Memory Distributed Storage System. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC)*. 65–79.

[5] Sumeet Bajaj and Radu Sion. 2013. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 26, 3 (2013), 752–765.

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–26.

[7] David Brumley and Dan Boneh. 2005. Remote Timing Attacks Are Practical. *Computer Networks* 48, 5 (2005), 701–716.

[8] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API Is a Bad Untrusted RPC Interface. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 253–264.

[9] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*. 857–874.

[10] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[11] Yu Ding, Ran Duan, Long Li, Yueqiang Cheng, Yulong Zhang, Tanghui Chen, Tao Wei, and Huibo Wang. 2017. Poster: Rust SGX SDK: Towards Memory Safety in Intel SGX Enclave. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2491–2493.

[12] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. 2015. M2R: Enabling Stronger Privacy in MapReduce Computation. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*. 447–462.

[13] Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont. 2019. Everything You Should Know About Intel SGX Performance on Virtualized Systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3, 1, 1–21.

[14] Josep Domingo-Ferrer, Oriol Farràs, Jordi Ribes-González, and David Sánchez. 2019. Privacy-Preserving Cloud Computing on Sensitive Data: A Survey of Methods, Products and Challenges. *Computer Communications* 140 (2019), 38–60.

[15] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning (ICML)*.

[16] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proc. VLDB Endow.* 13, 2 (2019), 169–183.

[17] Jason Evans. 2006. Jemalloc. In *Proceedings of the 2006 BSDCan Conference (BSDCan)*.

[18] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the PENGLAI Enclave. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 275–294.

[19] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic Analysis: Concrete Results. In *Proceedings of the Cryptographic Hardware and Embedded Systems (CHES)*. 251–261.

[20] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Ph.D. Dissertation. Stanford University. Advisor(s) Boneh, Dan.

[21] Sanjay Ghemawat and Paul Menage. 2009. *Tcmalloc: Thread-Caching Malloc*.

[22] Wolfram Gloger. 2006. *Ptmalloc Library*.

[23] Oded Goldreich. 1987. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*. 182–194.

[24] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*. 218–229.

[25] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[26] 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.

[27] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. 1993. Improving the Cache Locality of Memory Allocation. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI)*. 177–186.

[28] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptol. ePrint Arch.* 2016 (2016), 204.

[29] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*.

[30] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP)*. 191–205.

[31] Intel. 2017. *Memory Encryption Technologies Specification*. Retrieved June 30, 2022 from https://software.intel.com/content/dam/develop/external/us/en/documents-tps/multi-key-total-memory-encryption-spec.pdf

[32] Intel. 2018. *Intel Software Guard Extensions (Intel SGX) Developer Guide*. Retrieved June 30, 2022 from https://software.intel.com/content/www/us/en/develop/download/intel-software-guard-extensions-intel-sgx-developer-guide.html

[33] Intel. 2020. *10th Gen Intel Core Processor Families Datasheet, Vol. 1*. Retrieved June 15, 2022 from https://www.intel.com/content/www/us/en/products/docs/processors/core/10th-gen-core-families-datasheet-vol-1.html

[34] Intel. 2021. *3rd Gen Intel Xeon Scalable Processors with Intel SGX*. Retrieved June 30, 2022 from https://www.intel.com/content/www/us/en/products/docs/processors/xeon/3rd-gen-xeon-scalable-processors-brief.html

[35] Jianyu Jiang, Xusheng Chen, TszOn Li, Cheng Wang, Tianxiang Shen, Shixiong Zhao, Heming Cui, Cho-Li Wang, and Fengwei Zhang. 2020. Uranus: Simple, Efficient SGX Programming and Its Applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 826–840.

[36] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. 2020. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 971–985.

[37] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-Memory Key-Value Storage with SGX. In *Proceedings of the 14th EuroSys Conference 2019*. 1–15.

[38] UC Berkeley RISE Lab. 2022. *Opaque Benchmarking*. Retrieved June 30, 2022 from https://mc2-project.github.io/opaque-sql-docs/src/benchmarking/benchmarking.html

[39] UC Berkeley RISE Lab. 2022. *Supported Functionalities in Opaque*. Retrieved June 30, 2022 from https://mc2-project.github.io/opaque-sql-docs/src/usage/functionality.html

[40] Do Le Quoc, Franz Gregor, Jatinder Singh, and Christof Fetzer. 2019. SGX-PySpark: Secure Distributed Data Analytics. In *The World Wide Web Conference*. 3564–3563.

[41] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-Grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX security symposium (USENIX Security)*. 557–574.

[42] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs Over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD)*. 177–187.

[43] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[44] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64KB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. 234–251.

[45] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. Ghostrider: A Hardware-Software System for Memory Trace Oblivious Computation. *ACM SIGPLAN Notices* 50, 4 (2015), 87–101.

[46] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*. 359–376.

[47] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. 2015. Thermal Covert Channels on Multi-Core Platforms. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*. 865–880.

[48] Julian McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems (NIPS)*, Vol. 1. 539–547.

[49] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel Software Guard Extensions (Intel SGX) Software Support for Dynamic Memory Allocation inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*

2016. 1–9.

[50] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*.

[51] Sharad Mehrotra, Shantanu Sharma, Jeffrey Ullman, and Anurag Mishra. 2019. Partitioned Data Security on Outsourced Sensitive and Non-Sensitive Data. In *Proceedings of the IEEE 35th International Conference on Data Engineering (ICDE)*. 650–661.

[52] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*. 279–296.

[53] David Nassimi and Sartaj Sahni. 1979. Bitonic Sort on a Mesh-Connected Parallel Computer. *IEEE Trans. Comput.* 28, 01 (1979), 2–7.

[54] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*. 377–394.

[55] Kerim Yasin Oktay, Sharad Mehrotra, Vaibhav Khadilkar, and Murat Kantarcioglu. 2015. SEMROD: Secure and Efficient MapReduce over Hybrid Clouds. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 153–166.

[56] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. 2019. SGX-LKL: Securing the Host OS Interface for Trusted Execution. *arXiv preprint arXiv:1908.11143* (2019).

[57] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database Using SGX. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*. 264–278.

[58] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2014. Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM. *IACR Cryptol. ePrint Arch.* 2014 (2014), 997.

[59] Marcelo Rosa and Keiko Fonseca. [n.d.]. Specification and Implementation of Reusable Secure Microservices D3. 2.

[60] Andrei Sabelfeld and Andrew C Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

[61] Jose Rodrigo Sanchez Vicarte, Benjamin Schreiber, Riccardo Paccagnella, and Christopher W Fletcher. 2020. Game of Threads: Enabling Asynchronous Poisoning Attacks. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 35–52.

[62] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS)*.

[63] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*. 38–54.

[64] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 3–24.

[65] Raja Sekar et al. [n.d.]. *Vega*. Retrieved June 30, 2022 from https://github.com/rajasekarv/vega

[66] AMD SEV-SNP. 2020. Strengthening VM Isolation with Integrity Protection and More. *White Paper, January* (2020).

[67] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. 2017. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework with Trusted Processors. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1211–1228.

[68] Youren Shen, Yu Chen, Kang Chen, Hongliang Tian, and Shoumeng Yan. 2018. To Isolate, or to Share? That Is a Question for Intel SGX. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*. 1–8.

[69] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking inside a Single Enclave of Intel SGX. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 955–970.

[70] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 121–135.

[71] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*.

[72] Rohit Sinha and Mihai Christodorescu. 2018. VeritasDB: High Throughput Key-Value Store with Integrity. *IACR Cryptol. ePrint Arch.* 2018 (2018), 251.

[73] Emil Stefanov, Marten Van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.

[74] Dave Tian, Joseph I Choi, Grant Hernandez, Patrick Traynor, and Kevin RB Butler. 2019. A Practical Intel SGX Setting for Linux Containers in the Cloud. In *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy*. 255–266.

[75] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)*. 645–658.

[76] Stefan Köpsell TUD. [n.d.]. Integrated Implementation of the Micro-Services for Distributed Big Data Applications D4. 4.

[77] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*. 1041–1056.

[78] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards Memory Safe Enclave Programming with Rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2333–2350.

[79] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. 2018. SGX-Perf: A Performance Analysis Tool for Intel SGX Enclaves. In *Proceedings of the 19th International Middleware Conference (Middleware)*. 201–213.

[80] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *Proceedings of the European Symposium on Research in Computer Security*. 440–457.

[81] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating Timing Side-Channel Leaks Using Program Repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 15–26.

[82] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP)*. 640–656.

[83] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (SFCS)*. 162–167.

[84] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 15–28.

[85] Kehuan Zhang, Xiaoyong Zhou, Yangyi Chen, XiaoFeng Wang, and Yaoping Ruan. 2011. Sedic: Privacy-Aware Data Intensive Computing on Hybrid Clouds. In *Proceedings of the 18th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 515–526.

[86] ChongChong Zhao, Daniyaer Saifuding, Hongliang Tian, Yong Zhang, and ChunXiao Xing. 2016. On the Performance of Intel SGX. In *Proceedings of the 13th web information systems and applications conference (WISA)*. 184–187.

[87] Mark Zhao and G. Edward Suh. 2018. FPGA-Based Remote Power Side-Channel Attacks. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*. 229–244.

[88] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proceedings of 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 283–298.