MAGE: Nearly Zero-Cost Virtual Memory for Secure Computation

Sam Kumar, David E. Culler, and Raluca Ada Popa
University of California, Berkeley

Abstract

Secure Computation (SC) is a family of cryptographic primitives for computing on encrypted data in single-party and multi-party settings. SC is being increasingly adopted by industry for a variety of applications. A significant obstacle to using SC for practical applications is the memory overhead of the underlying cryptography. We develop MAGE, an execution engine for SC that efficiently runs SC computations that do not fit in memory. We observe that, due to their intended security guarantees, SC schemes are inherently *oblivious*—their memory access patterns are independent of the input data. Using this property, MAGE calculates the memory access pattern ahead of time and uses it to produce a memory management plan. This formulation of memory management, which we call *memory programming*, is a generalization of paging that allows MAGE to provide a highly efficient virtual memory abstraction for SC. MAGE outperforms the OS virtual memory system by up to an order of magnitude, and in many cases, runs SC computations that do not fit in memory at nearly the same speed as if the underlying machines had *unbounded physical memory* to fit the entire computation.

1 Introduction

Secure Computation (SC) refers to cryptographic primitives that allow computation on encrypted data. An example of SC is secure multi-party computation, which allows two parties to perform a collaborative computation on private input data. Advances in cryptography over the years have steadily brought SC closer to practice. Recently, the use of SC in industry—in particular, homomorphic encryption (HE) and secure multi-party computation (SMPC)—has burgeoned. Companies offer services based on SC [12,19,27,38,46,75] (from secure collaborative learning to decentralized authentication and custody), large financial enterprises have added SC-based products [64], and cryptocurrencies secure billions of dollars with SC [91].

SC not only has high CPU overhead, but also requires high memory usage and, in the case of SMPC, high network usage. For example, a 64-bit integer, which requires only 8 B of memory when computing in plaintext, takes up 1 KiB of memory when using a garbled circuit (a type of SMPC). Efficiently running SC requires careful attention to not only CPU efficiency, but also memory and network demands.

CPU overheads can be reduced using hardware accelerators (e.g., GPUs, FPGAs) or specialized hardware (e.g., AES-NI). Network bandwidth continues to grow exponentially according to Nielsen's Law [62], and recent cryptographic improvements have relaxed network bandwidth demands for some SC protocols [10, 15]. But memory management remains problematic. Many recent cryptographic systems based on SC report that SC systems quickly run out of memory [66,79,94,95]. Once they do, the computation becomes prohibitively slow because the OS inefficiently swaps the large memory footprint to secondary storage. For example, the authors of Conclave [79] report that Obliv-C, an SMPC framework, can run a join on only 30,000 records before running out of memory, and state that SMPC "in practice only scales to a few thousand input records." Similarly, Senate [66], a secure collaborative analytics engine based on SMPC, can run a 16-party private set intersection on only 10,000 integers per party.

In this context, we address the research question: **Can SC execute efficiently when it does not fit in memory?** We answer this in the affirmative with our system MAGE.[1]

A natural starting point for MAGE is to specialize the OS page replacement policy to SC workloads. Unsurprisingly, this design suffers from some of the same pitfalls as classic virtual memory systems. Pages may not be fetched until a page fault occurs, requiring multiprogramming to keep the CPU busy [26]. Furthermore, classic page replacement algorithms perform poorly on some workloads [3], and a policy specialized to SC would likely be no different.

To mitigate these issues, we observe that SC is inherently *oblivious*. In particular, many SC protocols have *no data-dependent memory accesses*. This is because an SC protocol must not leak any information about the data contents via its memory access pattern. Our key insight in MAGE is that SC's inherent obliviousness allows us to calculate the access pattern for any computation *in advance* and use it to manage memory in a fundamentally more efficient way than classic OS paging. Unlike paging, which typically responds to page faults *reactively*, MAGE can *proactively* produce a memory management plan based on the program's memory access pattern. To highlight this distinction, we call our approach *memory programming* and the resulting plan a *memory program*. MAGE preplans the exact sequence of memory-storage transfers to issue at runtime, given a target memory consumption. Enabled by memory programming and the compute-to-memory ratio of SC workloads, MAGE executes certain SC programs that do not fit in memory at nearly in-memory speeds, as if memory were unbounded—in effect, virtual memory at nearly zero cost.

To understand the power of MAGE's preplanning based on SC's obliviousness, consider Belady's theoretically optimal

[1] MAGE stands for **M**emory-**A**ware **G**arbling **E**ngine.

paging algorithm (MIN) [3]. MIN, being a clairvoyant algorithm, is not realizable in the classic formulation of paging; it is typically used as a point of comparison to other realizable heuristics. But in the context of memory programming, MAGE can use MIN directly, because it knows the access pattern in advance. Memory programming allows MAGE to use an algorithm that is well-grounded in theory, instead of a heuristic (e.g., LRU or LFU) that sometimes performs poorly.

Yet memory programming also raises the bar for possible memory management strategies. For example, although MIN is an optimal paging algorithm, it unfortunately does not produce an optimal memory program. The reason is that MIN, like other paging algorithms, brings a page into memory only at the moment it is needed, thereby causing the program to stall while transferring the page. We can overcome this by leveraging SC's obliviousness once again, to prefetch according to the access pattern (i.e., with no false positives or false negatives) so that the program never stalls.

At its core, our approach to memory management is quite simple: MAGE optimizes storage bandwidth by evicting pages using MIN, and optimizes latency via prefetching and asynchronous eviction. Whereas classic paging algorithms typically rely on heuristics and empirical observations of what works well in practice [9], our memory programming approach is simple, well-grounded, robust, and performant.

While conceptually simple, the above strategy is challenging to instantiate efficiently. The reason is that MIN requires the entire memory access pattern to be materialized at once; it cannot be applied in a streaming fashion. Using Intel Pin [54], we found that an SC workload that runs in under an hour can issue *trillions* of memory accesses. Thus, materializing the access trace could require *terabytes* of space.

To address this, we leverage the strong precedent for using DSLs to specify SC programs [34, 78]. MAGE's planner represents the program as a bytecode recording higher-level operations specified in the DSL program. This is more succinct than recording individual memory accesses. For example, consider a program that adds two integers using garbled circuits, an SMPC protocol. Garbled circuits support only AND and XOR operations on encrypted *bits*, so the integer addition is ultimately decomposed into encrypted AND and XOR operations, each of which comprises many memory accesses. Yet, MAGE records the entire addition operation as a *single* entry in the bytecode. This works well because most of the addition operation's memory accesses are "uninteresting"—they are accesses to temporary variables (e.g., on the stack) that fit easily in memory, or to SC protocol state that should remain in memory for the entire program. The only consequential accesses for memory management—reading the two input integers and writing the output integer—are captured in the single entry MAGE records.

Once MAGE allows SC to efficiently expand beyond the physical memory limit, another limited resource (e.g., storage/network bandwidth or CPUs) of a single machine could become the bottleneck. Thus, we design MAGE to support *parallel* SC execution across multiple network flows, CPU cores, or machines. To do so, we observe that a distributed memory programming model allows SC to be parallelized in this way, without requiring MAGE's planner to reason about threads executing concurrently in the same address space.

Finally, we aim to support a variety of applications and protocols, including new ones that may emerge in the coming years. The challenge is that different SC protocols may be very different cryptographically and may support different operations efficiently. Fortunately, our memory programming approach allows us to build MAGE entirely in userspace on a Linux system, helping to make MAGE *extensible* to new applications and protocols. We carefully design a layered architecture for MAGE so that the DSL, bytecode, and interpreter can be extended for new SC protocols.

We implemented MAGE in C++ and apply it to two SC protocols: (1) garbled circuits, a type of SMPC, and (2) CKKS, a type of HE. We evaluated MAGE using 10 workloads, sized such that they do not fit in memory. MAGE outperforms the operating system's virtual memory for all 10 workloads, and outperforms it by 4–12× for 7 of them. Additionally, MAGE executes all 10 workloads at within 60% of in-memory speeds, and runs 7 of them at within 15% of in-memory speeds.

Even with our techniques, SC remains orders of magnitude slower than plaintext computation due to CPU and network overheads. That said, various applications like federated data analytics [1, 66], coopetitive machine learning [94], and privacy-preserving recommendation [63] *require* SC. Due to privacy constraints, running these applications in plaintext is not an option. By bringing memory management overhead for SC to nearly zero, MAGE helps make SC more practical and potentially enables more SC-based applications.

2 Secure Computation Background

2.1 Circuit Representation

As explained earlier, SC is inherently oblivious, meaning that any function f computed using SC cannot have data-dependent memory accesses. Thus, it is natural to describe the function f as a circuit C [13, 23, 37, 55]. C is a combinational circuit that accepts the arguments to f as inputs and produces the result of f applied to those arguments as its output. We write $C = (W, G)$, where W is a set of wires and G is a set of gates. Each *wire* represents a datum whose type is the unit of computation in the SC scheme (in most cases, it is the information stored in a single ciphertext). We denote the subset of W storing C's input as I, and the subset of W storing C's output as O. Each *gate* represents a computation supported by the SC scheme. We will typically assume that each gate has exactly one output wire, and that each $w \notin O$ is the input wire of at least one gate. Thus, $|W| = |G| + |I|$.

The particular data types represented in the wires and the types of supported gates depend on the particular SC scheme of interest. For the CKKS homomorphic encryption

scheme [16], each wire represents a *vector of real numbers* and each gate represents an element-wise *addition or multiplication* of those vectors. For garbled circuits [88], each wire represents a *single bit* and each gate represents a binary *AND operation or XOR operation* on those bits. Other SC schemes can be similarly formulated this way. Below, we explain CKKS and garbled circuits in greater depth.

2.2 CKKS Homomorphic Encryption

In the CKKS scheme [16], each ciphertext encodes a vector of real or complex numbers (stored with limited precision). Given ciphertexts $c_1 = \mathsf{Enc}(\vec{v}_1)$ and $c_2 = \mathsf{Enc}(\vec{v}_2)$, one can compute $\mathsf{Enc}(\vec{v}_1 + \vec{v}_2)$ and $\mathsf{Enc}(\vec{v}_1 \circ \vec{v}_2)$ (where $\circ$ is element-wise multiplication). The dimension of each vector depends on parameters chosen during key generation. Each ciphertext is assigned a level, which is a nonnegative integer. When performing the element-wise multiplication operation, both input ciphertexts must have the same level; the level of the output ciphertext is one less than the level of the inputs. Performing element-wise addition does not reduce the ciphertext level the way element-wise multiplication does. A ciphertext at level 0 cannot be used for element-wise multiplication. The maximum level of a ciphertext depends on the parameters chosen during key generation. While one can run a bootstrapping procedure to increase the level of a ciphertext, it is very expensive, and therefore not implemented by all libraries.

2.3 Garbled Circuits

Yao's garbled circuit protocol [88] (referred to simply as *garbled circuits*) allows two parties, called the *garbler* and the *evaluator*, to jointly compute a function f over their private inputs x_1 and x_2. The protocol requires f to be represented as a *boolean circuit* C. Unlike CKKS, there are no restrictions on C's depth. However, *both* parties have to execute the circuit.

First, the two parties run a protocol called *oblivious transfer* to obtain the (encrypted) wire values for their inputs without revealing their inputs. Then the garbler encrypts C in a special way called *garbling* to obtain $\widetilde{C}$, called a *garbled circuit*. The process of garbling is analogous to executing the circuit; a gate cannot be garbled until the (encrypted) values of both input wires are obtained, and garbling a gate produces, as a side effect, the (encrypted) value of the output wire. Then, the garbler sends $\widetilde{C}$ to the evaluator. The evaluator executes the circuit, executing each gate using the gate's garbled information in $\widetilde{C}$. Finally, the two parties communicate to decipher the plaintext values of the output wires.

If the parties would like to repeat the computation again with different inputs, they must re-garble C. It is insecure to reuse the same garbled circuit $\widetilde{C}$ with different sets of inputs.

More comprehensive explanations of garbled circuits, their underlying cryptography, and their state-of-the-art optimizations are available in other resources [6, 69, 86].

2.4 Efficiently Executing Circuits

In this section, we give background on existing techniques for efficiently executing cryptographic circuits. Although many of these techniques were developed for garbled circuits, they mostly apply to homomorphic encryption as well.

2.4.1 Naïve Baseline

Early garbled circuit systems, like Fairplay [55], JKS [41], and PSPW [65], allocate memory for all wires and store the entire garbled circuit in memory. The memory overhead is $\mathcal{O}(|W| + |G|)$. Because, for a well-formed circuit, $|G| + |I| = |W|$, this is equivalent to $\mathcal{O}(|W|)$.

2.4.2 Pipelining Garbling and Evaluation

After the garbler garbles a gate to include in $\widetilde{C}$, the garbler does not use that gate's garbled data. Similarly, once the evaluator evaluates a gate, it never again uses that garbled gate. Based on this observation, the HEKM system [37] operates without keeping the entire garbled circuit in memory, as follows. The garbler and the evaluator first agree on an order in which to execute the gates in C. Then, the garbler garbles each gate and streams the garbled gates to the evaluator, who evaluates the gates in the same order. In this way, all gates are garbled and evaluated, without materializing the full set of garbled gates at any one time. Because space is allocated for all wires in the circuit, the memory overhead is still $\mathcal{O}(|W|)$.

2.4.3 Reclaiming Wire Memory

When executing a circuit, one can discard the memory for a wire once all gates it feeds into have been executed. Only wires whose values have been computed and will be used in the future—the *live* wires—must be kept in memory. The KSS system [49] takes advantage of this by dynamically attaching a reference count to each wire; PCF [48] statically calculates when to reuse wire memory. Using interpretation techniques developed in PCF [48] and refined in Frigate [60], not even the plaintext circuit is materialized in memory. TinyGarble [73], EMP-toolkit [82] (for semi-honest SMPC), and EVA [23] also use variants of this technique. With this optimization, the memory demand is $\mathcal{O}(w)$, where w is the size of the largest set of live wires when executing the circuit. MAGE builds on this line of work by exploring how to efficiently swap to storage when w wires do not fit in memory.

3 Memory Overhead of Secure Computation

First, we discuss the memory overhead of SC. Then, we discuss the memory overhead for collaborative applications.

3.1 Analysis of the Memory Demand

The size of the circuit, for a computation, is proportional to the size of the computation. But in many cases, the memory demand is substantially smaller than the circuit size; only w wires need to be stored, where w is the size of the largest set of live wires when executing the circuit (§2.4.3).

In practice, circuits are often described in a programming language [34, 78] and gates are executed in the same order as the program is interpreted. In this execution order, live wires correspond to in-scope variables in the program. Thus, the memory usage of running an SC program has the same order of growth as running the same algorithm in plaintext.

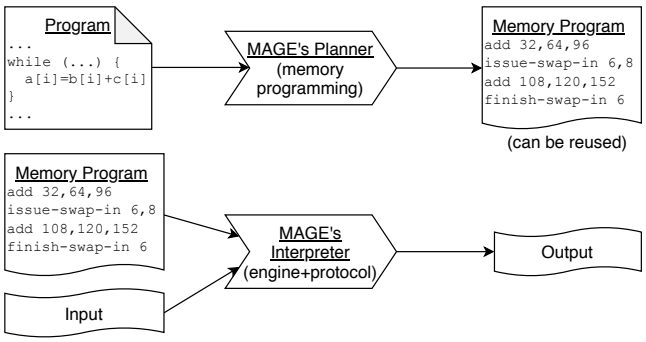


Figure 1: Overview of MAGE. It consists of two phases: planning (top) and execution (bottom)

The memory cost of SC lies in the constant factors. When executing a secure computation protocol, the wire values are encrypted. Thus, a key parameter is the expansion factor of the encryption. In garbled circuits using a 128-bit block cipher, including state-of-the-art optimizations (Point-and-Permute [2], Free XOR [47], Half Gates [90], and Fixed-Key Block Cipher [5, 31]), each wire value is 16 bytes. Each wire represents only 1 bit of plaintext, so this is a 128× expansion factor. For CKKS, ciphertexts at higher levels are larger than ciphertexts at lower levels. For the parameters we used in our evaluation, each ciphertext is hundreds of kilobytes and encodes a vector of dimension up to 4,096.

3.2 Scaling Collaborative Applications

SMPC supports *collaborative applications* over secret data, such as federated data analytics [1] and cooperative machine learning [59]. A common technique to reduce SMPC's overhead is to use SMPC in a *minimal way*. For example, some approaches aim to use SMPC for only a small part of the overall computation [1, 43, 53, 79, 94]. Others carefully choose algorithms that can be executed efficiently in SMPC or use approximations that incur less overhead [58, 59, 68]. But even with these approaches, the SMPC computation often has high memory demands [66]. Thus, it remains important to efficiently execute SMPC computations that do not fit in memory.

4 Overview of MAGE

SC workloads are oblivious by nature. Thus, MAGE can work out the program's memory access pattern in advance, and use this information to produce a memory management plan, called a *memory program*, tailored to the particular access pattern. Importantly, obliviousness is not merely an artifact of certain existing SC schemes; it is inherent to SC. Otherwise, an adversary could potentially infer information about secret data based on the memory access pattern.

To support this paradigm, MAGE's workflow has two phases, as shown in Fig. 1. An SC application is written in a DSL internal to C++. MAGE's planner unrolls the DSL code to produce a bytecode, and then performs transformations on the bytecode to produce a memory program. In MAGE, the memory program is a bytecode that includes *swap directives* describing when to transfer data between storage and memory. Finally, the memory program is given to MAGE's interpreter, which executes it using the SC protocol.

For multi-party protocols, the parties run separate instances of MAGE's interpreter. In the case of garbled circuits, garbled gates are streamed from the garbler to the evaluator, as described in §2.4.2. Both the garbler and evaluator use MAGE to follow a memory program and run with constrained memory.

Our approach of including swap directives in the memory program relies on the planner knowing how much memory will be available at runtime. An alternative approach is for memory programs to be agnostic to the amount of available memory. This would add runtime overhead, as MAGE's interpreter would need to decide which pages to evict. In contrast, our approach moves this overhead to the planning phase, keeping the execution phase as lightweight as possible.

4.1 Address Translation in MAGE

The application programmer should not have to manage paging, so it is natural to write DSL programs in a virtual address space that is, in effect, infinitely large. Central to designing MAGE is deciding at which point in Fig. 1 to translate this address space into a physical address space that fits in RAM.

One possibility (which MAGE does not use) is to perform address translation at runtime, using standard operating system mechanisms for prefetching and address translation. At runtime, swap directives in the memory program would ask the operating system to page parts of the virtual address space out to storage or in to RAM. Unfortunately, the existing way for a Linux process to do this—the `madvise` system call—is too limited. As of Linux 5.10, pages brought into RAM using the `MADV_WILLNEED` hint are not mapped in the page table, so a minor page fault is incurred on the first subsequent access. Similarly, the `MADV_PAGEOUT` hint merely marks pages as inactive; it does not swap out pages immediately.

In contrast, MAGE does not rely on OS address translation for demand paging. MAGE's engine moves data between memory and storage via explicit I/O operations, so that its resident set size never exceeds the available RAM. At the surface, this is similar to buffer management in a DBMS. But unlike a DBMS, MAGE's planner can be viewed as solving an address translation problem in advance. The DSL variables declared by the programmer exist in a *MAGE-virtual address space*, and the final memory program output by the planner references data (i.e., wire values) in a *MAGE-physical address space* that fits within RAM. MAGE's planner creates these address spaces and performs their translation in software during the planning phase. It includes swap directives in the memory program so that the interpreter does not run out of RAM.

To avoid confusion, we will refer to the addresses created by the OS and sent over the memory bus as *OS-virtual addresses* and *OS-physical addresses*. At runtime, MAGE's interpreter stores the program's memory in an array, and each MAGE-physical address in the memory program is treated as an index into this array. Thus, MAGE-physical addresses roughly correspond to the OS-virtual addresses of MAGE's interpreter.

MAGE's approach to address translation has several advantages. First, in contrast to an `madvise`-based approach, MAGE's planner has nearly complete control over when pages are brought into memory and evicted to storage. Second, by translating addresses in the planner, MAGE avoids address-translation-related overheads at runtime. In contrast, relying on OS address translation would mean minor page faults, page table updates, and TLB invalidations at runtime.

MAGE's approach also has a few drawbacks, however. First, the planning phase takes longer because MAGE's planner must translate all addresses in software. Second, memory programs are considerably larger because they must contain not only swap directives, but also a copy of the program translated to operate on MAGE-physical addresses. In particular, the memory program's length is proportional to the program's execution time because a variable local to a function or loop could be assigned different physical addresses each time the function is called or on each iteration of the loop.

Overall, we felt that the advantages of this design outweighed its drawbacks. Longer planning times seemed reasonable because planning can happen offline and the resulting memory program can be used repeatedly. The larger memory program size was an acceptable tradeoff because MAGE's planner materializes an unrolled form of the program anyway to run Belady's algorithm. Meanwhile, MAGE's planner is afforded nearly full control of page eviction and replacement and MAGE's runtime overheads remain relatively small.

4.2 MAGE's Bytecode Representation

Recall that MAGE's planner expresses the program as an unrolled (branch-free) bytecode, and performs transformations on it to compute the memory program bytecode. What operations should the bytecode instructions support?

One possibility would be for the bytecode to describe low-level operations similar to those supported by a CPU, excluding control flow instructions. Unfortunately, such a bytecode includes the raw memory trace of the program, which, as discussed in §1, can be impractically large.

One alternative, used by PCF [48] and Frigate [60][2] (but not MAGE), is to have each instruction correspond to a gate in the circuit C being executed. This approach would require a *protocol driver* in MAGE's interpreter that executes each gate using the SC protocol. To understand why this is inefficient, consider garbled circuits, for which gates are binary and wires represent bits. The programmer specifies the circuit in terms of operations on high-level types such as integers, which are then compiled into bit-level operations. Thus, each time the program performs a high-level operation (e.g., adding two integers), the same subcircuit (e.g., describing integer addition in terms of binary gates) is repeated in the bytecode.

To eliminate this repetition, MAGE has each instruction describe a high-level operation directly. This requires not only a *protocol driver*, but also an *engine* in MAGE's interpreter that expands each instruction into the relevant subcircuit at runtime. MAGE's planner does not need to materialize the subcircuits because wires internal to the subcircuits are very short-lived and therefore can be ignored.

[2]Unlike MAGE, these systems also include control flow operations.

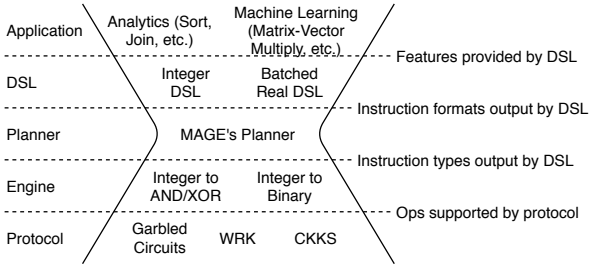


Figure 2: MAGE's envisioned ecosystem, with planning as the narrow waist

4.3 MAGE's Ecosystem and its Extensibility

An important consideration in MAGE's design is to be applicable to a range of SC protocols. For example, garbled circuits and homomorphic encryption (CKKS) have quite different computation models, yet we show how MAGE captures both. MAGE's envisioned ecosystem can be understood as a set of layers with a narrow waist, as shown in Fig. 2. The narrow waist is MAGE's planner; MAGE's core planning algorithms can be used with a variety of applications and interpreters.

MAGE's interpreter has two layers. The upper layer, called the *engine*, decomposes each instruction into a subcircuit of gates supported by the target SC protocol (§4.2). The lower layer, called the *protocol driver*, evaluates gates with the SC protocol. For example, when using a protocol that supports only binary AND and XOR operations (e.g., garbled circuits), one must use an engine that decomposes each instruction into a circuit of AND and XOR gates. In contrast, when using a protocol that supports all types of binary gates (e.g., TFHE [17]), one can use an engine that uses all types of gates.

One must choose compatible implementations at each layer. For example, once one has selected an SC protocol, one should choose an engine that executes each instruction using operations supported by that protocol. Then, one should select a DSL that outputs instructions that the chosen engine understands. Finally, one must write the application in that DSL.

MAGE's planner, however, is universally compatible, allowing it to be the "narrow waist" of the ecosystem. The first reason is that MAGE's planner does not have to understand what each instruction *does*, only what memory it accesses. Thus, even if a new instruction is introduced into a DSL, extending a header file to specify its format (which includes which fields are memory addresses) is enough for the planner to understand that instruction. The second reason is that MAGE's planner does not introduce any new instructions except for swap directives, which all engines understand. Thus, if an engine understands the instruction types output by MAGE's DSL, then the engine will also be able to interpret the planner's output (i.e., the memory program).

A number of frameworks and DSLs for SC [34, 78] aim to make it easier for non-SC-experts to use SC. In contrast, MAGE is an efficient SC execution engine; its DSLs are not necessarily geared toward non-experts, do not optimize the resulting circuit, and might expose low-level SC operations. We discuss how these frameworks fit into Fig. 2 in §9.

5 MAGE's Engine

MAGE's execution engine is an interpreter for the final memory program. First, it allocates an array to store the program's data. Each MAGE-physical address is an index into this array. To execute an instruction, MAGE reads the instruction's arguments from this array, makes calls to the protocol layer to compute the output, and writes the output back to the array. Each instruction in the memory program references its input and output data directly by MAGE-physical address; the engine sees no MAGE-virtual addresses. Some instructions, such as those requesting pages to be transferred between storage and memory, are handled directly by the engine, without calling the protocol. We call such instructions *directives*.

5.1 Parallel/Distributed Engine

SC is resource-intensive, so it is natural to scale SC by executing the protocol in a distributed fashion across *multiple CPU cores* or *multiple machines*. The multiple-machine case is useful to overcome resource constraints associated with a single machine such as limited CPU cores, limited storage I/O, or, in the case of SMPC, limited network bandwidth. This is different from having multiple parties in SMPC. Here, we are parallelizing a single trust domain—for example, a single logical party in SMPC may execute using multiple machines.

MAGE's engine supports distributed execution across multiple *workers*. Each worker is a thread of computation, running MAGE's engine, operating on its own memory region (a MAGE-physical address space). Workers differ from OS processes as follows: (1) each worker contains exactly one thread, (2) workers are not necessarily isolated by hardware such as an MMU—multiple workers in a MAGE computation could, in principle, run within the same process, and (3) memory is statically partitioned among the workers.

MAGE's planner does not automatically infer how to parallelize the computation. Rather, the programmer writes DSL code in a distributed memory model, explicitly indicating asynchronous network operations to transfer data among the different workers. The resulting memory program bytecode contains *network directives* that the engine interprets. Similarly, the protocol driver must be written to function properly when the computation is distributed over multiple workers.

Programs for MAGE are parameterized by the Worker ID. MAGE's planner is run once *for each worker*. To generate the memory program for a worker, the planner processes only the accesses for that worker—it does not need to consider other workers' accesses, because each worker can only access its own memory region. Thus, the workers' memory programs can be generated independently and in parallel.

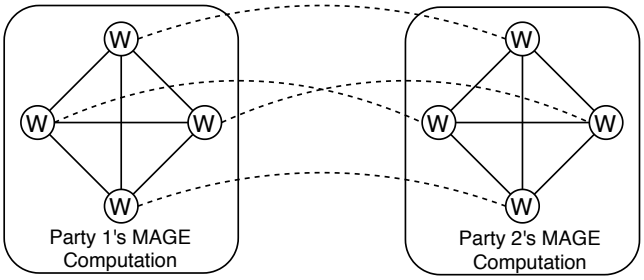


Figure 3: Example of distributed SMPC with MAGE. Workers are denoted as circles with W. Solid lines indicate connections managed by MAGE's engine; dashed lines indicate connections managed by the protocol driver

Using a distributed memory model provides two benefits. First, it allows MAGE to be agnostic to whether workers are placed on a single machine or across multiple machines. Second, it guarantees that the access pattern for each region of memory consists of a single well-defined sequence, simplifying planning. To ease the difficulty of explicitly specifying network transfers, one can build easier-to-use DSL libraries for common communication patterns (e.g., our implementation provides a `ShardedArray<T>` abstraction).

5.2 Distributed SMPC

Some SC protocols, like SMPC, require interaction over the network between mutually distrusting parties. For such protocols, each party runs a separate MAGE computation, with its own set of workers. Whereas the MAGE engine handles *intra-party communication* between workers in the same party, the protocol implementation handles *inter-party communication* among workers in different parties. The inter-party topology is up to the protocol driver; our protocol driver for garbled circuits uses a one-to-one inter-party topology (Fig. 3).

6 MAGE's Planner

Our memory programming approach is to calculate the memory access pattern in advance and use it to preplan memory management. One can potentially preplan the following:

- **Placement.** How should we divide up a circuit into pages?
- **Ordering.** In what order should we evaluate the gates in the SC circuit to result in the best memory behavior?
- **Scheduling.** When should pages that will be used in the future be swapped in from storage?
- **Replacement.** How should we choose pages to evict when making room for pages from storage?

MAGE produces an approximate solution, using a heuristic for placement and optimizing scheduling and replacement. Note that MAGE does not optimize ordering; it evaluates gates in the order implicit in the DSL program for the circuit.[3]

6.1 Organization of MAGE's Planner

We organize MAGE's planner into stages (Fig. 4):

[3] Optimizing ordering may be NP-hard [76]. A system that does so would be very powerful—for example, it would automatically block a loop join or tile a matrix multiplication. It is beyond the scope of this work.

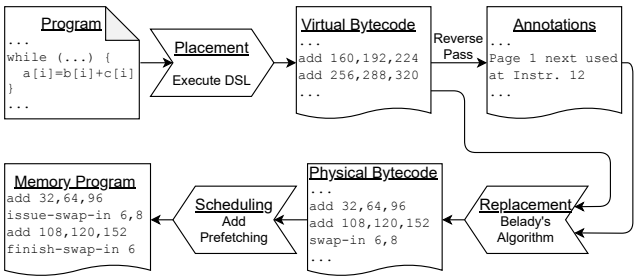


Figure 4: MAGE's planner's workflow, with its three stages

1. **Placement.** This stage accepts a DSL program and organizes wires into MAGE-virtual pages. It outputs instructions referencing wires by MAGE-virtual address.
2. **Replacement.** This stage adds instructions to swap pages to/from storage, deciding which pages to evict. It outputs instructions referencing wires by MAGE-physical address.
3. **Scheduling.** This stage moves swap instructions within the instruction stream and relocates wires to mask the latency of moving data between memory and storage.

For a parallel/distributed program, MAGE's planner is invoked separately for each worker, with separate MAGE-virtual and MAGE-physical address spaces. Network directives in the program transfer data among those address spaces.

MAGE's planner does not benefit from MAGE's memory programming techniques, so it is important that planning does not consume an unreasonable amount of memory. We keep the planner's memory usage lightweight by (1) writing/reading the intermediate bytecodes to/from files instead of keeping it all in memory, (2) designing the DSLs to be lightweight, and (3) keeping track of pages instead of individual bytes.

6.2 MAGE's First Stage: Placement

MAGE's placement module is, in effect, a page-aware memory allocator for the DSL. It unrolls the DSL, allocating space for each variable and intermediate value in the MAGE-virtual address space. It outputs a bytecode for the program in which each variable is referenced by its MAGE-virtual address.

6.2.1 Unrolling the DSL Code

MAGE's DSLs are internal to C++. This means that the DSL is a set of convenient C++ APIs to specify the program's behavior, often involving operator overloading. The program is specified as a C++ function that uses these APIs.

Fig. 5 shows a program that solves Yao's Millionaire's problem [87]. `Integer<width>` describes an Integer datum with the specified width in bits. `Bit` is an alias for `Integer<1>`.

MAGE's planner does not parse the DSL program's source code or manipulate its AST. Instead, it simply calls the C++ function containing the DSL program. As the DSL code executes, it produces a bytecode describing the computation. For example, the overloaded `+` operator for `Integer` emits an `Add` instruction in the output bytecode; it does not actually add integers using secure computation. Each output instruction references its operands by MAGE-virtual address. Thus, the DSL (e.g., the `Integer` class) calls MAGE's placement module to allocate memory in the MAGE-virtual address space for intermediate results, including those stored in variables.

```
void millionaire(const ProgramOptions& args) {
    Integer<32> alice_wealth, bob_wealth;
    alice_wealth.mark_input(Party::Garbler);
    bob_wealth.mark_input(Party::Evaluator);
    Bit result = alice_wealth >= bob_wealth;
    result.mark_output();
}
```

Figure 5: Example code in an Integer-based DSL internal to C++ to solve Yao's Millionaire's problem

For example, see Fig. 5. On the `mark_input` and `>=` operations, an allocation request is made to MAGE's placement module to obtain a MAGE-virtual address, and an instruction is emitted to perform that operation (obtain input or integer comparison) and store the result at that MAGE-virtual address. Once an `Integer`'s destructor is called, or if an `Integer` is reassigned to a new MAGE-virtual address, a deallocation request is made to MAGE's placement module for the MAGE-virtual address previously held by that `Integer`.

For a parallel/distributed program, the worker ID and total number of workers are provided via the `ProgramOptions` structure. The C++ code can branch on these variables, to have each worker operate differently and exchange data appropriately to perform the parallel/distributed computation.

Each `Integer` object contains only the MAGE-virtual address of its contents; other attributes, such as width, are template arguments and do not consume memory. Thus, `Integer`s and other DSL-provided data types are typically smaller than the encrypted data items they represent. For example, a 32-bit integer encrypted for the garbled circuit protocol is 1 KiB in size, whereas an `Integer<32>` object used during planning is just 8 B (a single MAGE-virtual pointer). This helps keep the memory cost of the planning phase small.

6.2.2 Memory Allocation Strategy

When MAGE's placement module allocates memory for a variable, it ensures that the variable is contained in a single MAGE-virtual page; a variable must never straddle two pages. The reason is that two adjacent MAGE-virtual pages may not be adjacent in the OS-virtual address space at runtime.

A key issue in designing the placement module's memory allocator is internal fragmentation [25, 67]. Some fragmentation, which we call *classic fragmentation*, arises from the inability to pack variables onto pages (e.g., part of a page's space cannot store any variable). Another type of fragmentation, which we call *effective fragmentation*, arises from the page's lifetime exceeding some of the variables it stores; if even one wire on a page is alive, the entire page remains alive.

To reduce classic fragmentation, MAGE's placement stage uses techniques from slab allocators [8]. Each page contains only variables of a particular size. When a variable goes out of scope in the DSL, its "slot" in its page is marked as free.

When a space for a variable must be allocated, MAGE's placement module look for a free slot in a page containing variables of that size; if no such pages have free slots, it allocates a new page for variables of that size. The slab size is one MAGE-virtual page. This ensures that no variable will straddle a page boundary. Just as in slab allocators, some leftover space at the end of a page may be unusable, but this can be controlled by tuning the page size. Unlike slab allocators, MAGE's placement module does not preserve object state across allocations.

To reduce effective fragmentation, MAGE's placement stage uses the following heuristic when allocating memory for a variable. If multiple pages, for the specified variable size, have free slots available, then MAGE uses the candidate page with the fewest free slots. This allows the number of live pages to decrease if the number of live variables decreases, by giving a chance for all variables on a page to die.

6.3 MAGE's Second Stage: Replacement

We apply Belady's MIN algorithm [3]. MIN is theoretically optimal in the number of SWAP-IN operations, but it does not minimize the number of swap operations if SWAP-OUT operations are also considered. The reason is that only dirty pages need to be written back to storage (i.e., "swapped out"). Minimizing the number of swaps when taking this into account is NP-hard [28]. Regardless, MIN produces a solution with at most $2\times$ as many swaps as the theoretical optimum,[4] so it is useful in MAGE's replacement stage.

To use MIN, we first make a backward pass over the program to determine, each time a page is used, the time (instruction ID) at which it is used next. Then we make a forward pass over the program, using the annotated next use time to determine which page to swap out. This requires us to maintain a priority queue of resident pages, so that we can quickly identify which one's next use is farthest in the future. Each instruction, even if its arguments are already resident, requires us to also perform a `decrease_key` operation on the priority queue to adjust pages' next use time. Therefore, if N is the number of instructions and T is the number of pages that fit in memory, applying Belady's MIN algorithm is $\mathcal{O}(N \log T)$.

This stage outputs an instruction stream that contains swap directives and references wires by MAGE-physical address. To support this, MAGE's planner maintains a data structure that maps MAGE-virtual page numbers to MAGE-physical frame numbers, similar to a page table.

When planning a parallel/distributed program, the planner must be careful to not steal a page that is currently being used for network I/O. Thus, MAGE's replacement phase reads the network directives to infer the outstanding asynchronous network operations. When stealing pages, it issues *network barrier* directives, as necessary, to ensure that the engine waits for the relevant network I/Os to complete.

[4] This occurs in the worst case where it evicts only dirty pages, but there is an optimal solution that evicts the same number of clean pages.

6.4 MAGE's Third Stage: Scheduling

We introduce a parameter ℓ called the *lookahead*. To prefetch data, MAGE's scheduling algorithm attempts to move SWAP-IN directives ℓ instructions earlier in the instruction stream. However, this does not work if one of the ℓ intervening instructions uses the page frame into which we are bringing in data. We solve this by budgeting B extra physical page frames, called the *prefetch buffer*; the replacement stage is now run with a capacity of $T - B$ frames, not T frames. Data is brought asynchronously into a free slot in the prefetch buffer. Only when it is finally needed is it copied from the prefetch buffer into its destination physical page frame. Instead of SWAP-IN directives, the memory program contains ISSUE-SWAP-IN directives, which initiate the transfer of a page into memory, and FINISH-SWAP-IN directives, which block execution until a swap operation has completed. Ideally, swap operations will be scheduled such that FINISH-SWAP-IN never blocks, but it serves as an important fallback to prevent old/corrupt data from being used if the transfer is unpredictably delayed.

We use the prefetch buffer similarly to swap out pages. The page to be swapped out is copied into a free slot in the prefetch buffer and then swapped out to storage with an ISSUE-SWAP-OUT directive while execution of subsequent instructions continues. Unlike SWAP-IN operations, there is no clear deadline by which the write to storage must complete. Thus, we delay issuing a FINISH-SWAP-OUT directive for as long as possible; we only issue it when allocating a slot in the prefetch buffer fails. In such a situation, we identify the oldest ISSUE-SWAP-OUT operation, issue the FINISH-SWAP-OUT directive for it, and reclaim its page in the prefetch buffer.

One could eliminate the copying of pages to/from the prefetch buffer by rewriting future instructions. We did not implement this optimization because it would introduce additional complexity and MAGE performs well without it.

A natural question is how large B must be. SSDs have bandwidths less than 10 GB/s and latencies that are usually less than 1 ms. Based on these measurements, Little's Law gives: $B = 10\text{ GB/s} \cdot 1\text{ ms} = 10\text{ MB}$. For server-class machines, this is $< 1\%$ of physical memory. In practice, we use 16–32 MiB to account for burstiness/queuing, still only a small fraction of available memory. Thus, MAGE's scheduling promises to mask storage latency with only a small memory penalty.

7 Implementation

We implemented a prototype of MAGE in C++, including support for two protocols: garbled circuits and CKKS. Using `cloc`, we found that our implementation is $\approx 11{,}000$ lines of code, excluding comments and blank lines, broken down as follows: $\approx 2{,}800$ for common libraries used throughout MAGE (e.g., data buffering for I/O, configuration file parsing, etc.); $\approx 1{,}300$ for MAGE's planner; ≈ 900 for protocol drivers (not including the underlying cryptography); $\approx 1{,}000$ for MAGE's DSLs and libraries for those DSLs (e.g., for sharding data); $\approx 1{,}100$ for MAGE's engines; $\approx 1{,}600$ for SC

programs written in MAGE's DSLs, used for testing and evaluating MAGE; $\approx 1,900$ for the underlying cryptography for garbled circuits, much of which is based on EMP-toolkit [82]; and ≈ 400 for in-progress (not yet complete) support for a third protocol. We build MAGE using `clang++` version 10.0.0 with the optimization flags `-Ofast -march=native`. MAGE runs as a Linux process, with no changes to kernel code.

7.1 MAGE's Interpreter

Engine. The `Engine` class implements common functionality for the engine layer, including support for directives. It establishes pairwise TCP connections among workers within a single party, to support network directives. Swap directives are implemented using the `aio` facility provided by the kernel (not to be confused with POSIX `aio`); the swap file/device is `open`ed with the `O_DIRECT` flag. MAGE engines are implemented as class templates that extend (inherit from) the `Engine` class. The protocol driver class is provided to the engine as a template argument, so the engine can make calls to it. We avoided using virtual functions for this, as their overhead can be significant (e.g., for free XORs).

Protocol Driver. The protocol driver exposes the SC protocol's native operations to the engine as a set of methods. When the engine invokes these methods, it provides pointers to data to operate on, stored in a large array representing the MAGE-physical address space. The protocol driver specifies the type of entries in the engine's array, in effect dictating what each MAGE-physical address actually corresponds to for its protocol (plaintext bits, ciphertext bytes, etc.), and provides a plugin to the DSL so it can allocate MAGE-virtual memory accordingly. The protocol driver must not store pointers to dynamically allocated memory in the array. The reason is that the engine swaps out only the contents of the array, not including any dynamically-allocated memory it points to. In addition to the SC protocol's cryptographic routines, the driver manages all protocol-specific operations. This includes sharing protocol-specific state among workers within a party, obtaining input data, writing output data, and managing intra-party communication where necessary (e.g., sending garbled gates from the garbler to the evaluator).

7.2 Extending MAGE with New Protocols

To extend MAGE with a new protocol, one must, at minimum, write a protocol driver to support it. If the operations exposed by the new protocol driver are identical to those exposed by an existing protocol driver, then one can use the same engine that works with the existing protocol. Otherwise, one must implement a new engine or modify an existing engine. This involves deciding which instruction types the new engine will be compatible with. If the supported instruction types differ from what existing DSLs produce, then one may have to implement a new DSL or modify an existing DSL.

We implemented protocol drivers for garbled circuits and CKKS. Garbled circuits and CKKS support different operations, so we implemented a separate DSL (Integers vs. Batches) and engine (AND-XOR vs. Add-Multiply) for each protocol. This conveniently allows us to showcase MAGE's ability to support different implementations of each layer. That said, it is not uncommon for related SC protocols to expose similar interfaces. For example, the WRK protocol [83, 84] exposes the same interface as garbled circuits (AND-XOR), so support for WRK, if added, could reuse our Integer DSL and AND-XOR engine.

7.3 Garbled Circuit Protocol Driver

For garbled circuits, wires have uniform size, so we allow MAGE address spaces to be wire-addressed; the DSL is unaware of the size of wires in bytes. Some subcircuits used by the AND-XOR engine are based on those used by Obliv-C [89]. Our garbled circuit driver uses cryptographic kernels from EMP-toolkit [82]. We implement oblivious transfer (OT) using multiple background threads. Concurrently with our work, EMP-toolkit was updated to use the MiTCCRH hash function [31]; our implementation is based on an older version of EMP-toolkit based on fixed-key AES [5]. When we compare MAGE to EMP-toolkit in §8, we use the older version of EMP-toolkit so the comparison is fair. This is not a limitation of MAGE; our driver could be changed to use MiTCCRH.

7.4 CKKS Protocol Driver

CKKS ciphertexts vary in size depending on their level, so for CKKS' DSL and engine, MAGE address spaces are byte-addressed. The protocol driver provides a plugin to the DSL describing the particular wire sizes in bytes. It uses the CKKS implementation in Microsoft SEAL [71]. We chose parameters for CKKS that allow a multiplicative depth of 2. A challenge was that SEAL ciphertext objects contain pointers and dynamically-allocated memory. MAGE cannot swap such objects to storage (see §7.1). Thus, TE protocol driver serializes ciphertexts using SEAL's built-in serialization methods when they are not in use; each operation (e.g., add, multiply) deserializes the arguments, computes the result, and then serializes the result. We quantify the cost of serialization in §8. This overhead is not fundamental; CKKS ciphertexts could be implemented as flat buffers, or homomorphic operations could be implemented to operate directly on serialized ciphertexts.

After a multiplication, CKKS ciphertexts are typically relinearized and rescaled before the next multiplication. But if two products are added (e.g., $ab + cd$), one can perform relinearization once for the overall result instead of for each multiplication separately (e.g., ab and cd). MAGE's DSL supports this optimization, which is crucial to achieve good performance on **rstats** and the linear algebra workloads.

8 Evaluation

8.1 Workloads

We now establish a set of SC workloads for our evaluation. Garbled circuits and CKKS support different operations—bitwise operations for garbled circuits, and add-multiply circuits of low multiplicative depth for CKKS—so we design

separate workloads for each protocol. These workloads are data-intensive "kernels" that may be used as part of larger SC applications. We discuss larger SC applications in §8.8.

8.1.1 SMPC Collaborative Applications

One application of SMPC is federated data analytics [66, 79]. Aggregations (`GROUP BY` operations) and joins are particularly memory-intensive. A federated data analytics system may express equi-joins as set intersections (SI) and aggregations as set unions (SU), both of which can be implemented by merging sorted lists [66]. This inspires our first benchmark, **merge**: *merging sorted lists of records*. In some cases, the input lists may not be already sorted. This inspires our second benchmark, **sort**: *sorting a list of records*. For joins other than equi-joins, the system must fall back to a classic loop join. This is our third benchmark, **ljoin**: *loop join*. For concreteness, we assume that each record is 128 bits long, and that the first 32 bits are the key used for sorting or joining; the problem size n is the number of records per party.

Privacy-preserving machine learning applications inspire our fourth benchmark, **mvmul**: *matrix-vector multiply with 8-bit integers*. A recent proposal for secure neural network inference, XONN [68], suggests *binarizing* the neural network. This inspires our fifth benchmark, **binfclayer**: *binary fully-connected layer*. It consists of a series of XNOR and PopCount operations similar to multiplying a binary matrix by a binary vector, followed by a binary activation function. For simplicity, we do not include batch normalization.

8.1.2 CKKS Homomorphic Encryption

We restrict ourselves to workloads for which CKKS is efficient—workloads that can be expressed as arithmetic circuits of low multiplicative depth. The sixth workload is **rsum**: *sum of a list of real numbers*, which requires no multiplications. The seventh workload is **rstats**: *computing the mean and variance of real numbers*, which requires a multiplicative depth of 2. These represent simple data analytics workloads; the problem size n is the number of elements.

Our remaining workloads are inspired by machine learning and linear algebra. The eighth workload is **rmvmul**: *matrix-vector multiply with real numbers*. Finally, we consider two variants of matrix multiplication. The ninth workload is **n_rmatmul**: *matrix-matrix multiply with a naïve nested `for` loop*. The tenth workload is **t_rmatmul**: *tiled matrix-matrix multiply*. The problem size n is the length of one side of the matrix (also for **mvmul** and **binfclayer**).

8.1.3 Implementation of Workloads

For simplicity, our implementations of some of these workloads only support power-of-two sizes and power-of-two number of workers, but this is not a fundamental limitation of MAGE. Some workloads can, in principle, be optimized through streaming. For example, **rsum** could read each input one at a time, add the result to an accumulator, and then output the accumulator, instead of holding the entire input dataset in memory. We deliberately avoided such "optimizations," as they would not be possible if the workload were part of a larger computation whose intermediate results are held in memory. Thus, each workload operates in three non-overlapping phases: (1) the inputs are read into memory, (2) the computation is performed, materializing the output in memory, and (3) the output is written to a file.

For the parameters we chose, the CKKS scheme encrypts vectors of dimension 4096. Thus, each of our workloads for CKKS could be applied to 4096 instances of the problem in a SIMD fashion with no additional overhead. There are ways to use the 4096 slots in the vector to speed up a *single* problem, for example, by vectorizing matrix multiplication [42]. Our workloads, for simplicity, do not apply such techniques, but MAGE is not incompatible with them.

8.2 Empirical Methodology

We compare MAGE's performance to an upper bound and a lower bound. The upper bound, *OS Swapping*, is the speed when relying on the operating system's paging. The lower bound, *Unbounded*, is the speed when the entire computation fits in memory. We measure these three scenarios as follows:
1. Unbounded. MAGE's planner is run assuming enough memory to fit the program. Thus, MAGE's planner does not insert swap directives in the memory program. Finally, MAGE's engine executes the memory program outside of any `cgroup`.
2. OS Swapping. A memory program is generated in the same way as for the *Unbounded* solution. However, it is executed in a `cgroup` that limits physical memory to a fixed amount.
3. MAGE. MAGE's planner is run assuming a fixed physical memory capacity, minus the prefetch buffer and the interpreter's overhead. The resulting plan is run within a `cgroup` that limits physical memory to 1 GiB or 16 GiB, to ensure that the memory overhead fits in the limit.

Except where stated otherwise, we used `D16d_v4` instances on Microsoft Azure [57]. We chose this instance type for a few reasons. First, it has enough memory to fit the entire computation for most experiments, necessary for the Unbounded scenario. Second, it contains a local "temporary" SSD. We use it for swap space (one of its recommended uses [20]) and for the file containing the memory program. Third, it provides enough network bandwidth so as not to be a bottleneck for garbled circuits (we explore the WAN setting in §8.7).

We set MAGE's parameters as follows. For garbled circuits, we used a page size of 64 KiB, lookahead ℓ of 10,000 instructions, and prefetch buffer size B of 256 pages. For CKKS, we used a page size of 2 MiB, lookahead ℓ of 100 instructions, and a prefetch buffer size B of 16 pages. Because CKKS ciphertexts are large, we used a larger page size (slab size) than for garbled circuits to reduce external fragmentation. Additionally, we left an additional 32–64 MiB of memory unused, to accommodate the memory used by MAGE's interpreter.

8.3 Comparison to Existing Frameworks

We compare MAGE's garbled circuits performance to that of EMP-toolkit. Our goal is to demonstrate that MAGE's techniques do not limit the performance of garbled circuits

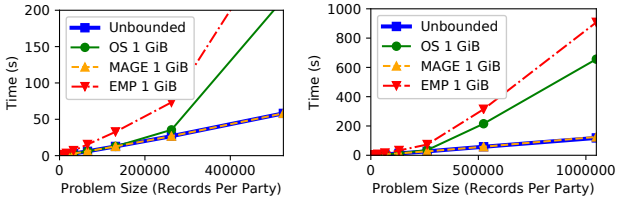


Figure 6: Comparison of MAGE and EMP-toolkit

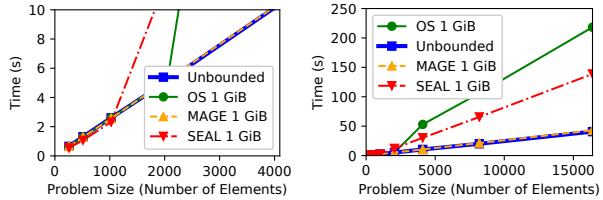


Figure 7: Comparison of MAGE and SEAL

compared to an existing system. We use **merge** for the comparison. We implemented **merge** in EMP-toolkit's DSL, and used EMP-toolkit's library for merging sorted arrays.

We discovered that EMP-toolkit is an order of magnitude slower than MAGE. This was because EMP-toolkit performs a separate invocation of OT extension, which involves a network round-trip, each time an Integer input is read for the evaluator. Our garbled circuits implementation for MAGE does not have this problem because it performs OTs in larger batches using background threads, regardless of the units by which the program reads the input. To eliminate this effect, we exclude the time to read the input, for both EMP-toolkit and MAGE, for this experiment only; we measured the time to merge the two arrays once they are materialized in memory.

We also compare MAGE's CKKS performance on **rstats** to a C++ program that uses SEAL directly. The main source of overhead in MAGE is the need to deserialize the input ciphertexts and serialize the output ciphertext, for each instruction.

The results are shown in Fig. 6 and Fig. 7. The graphs on the left are zoomed in to smaller problem sizes to show the point where memory demand exceeds available physical memory. "OS" refers to scenario 2 in §8.2; "EMP" and "SEAL" refer to those systems similarly running in a `cgroup`. EMP performs about 3× worse than OS when the problem fits in memory; when it does not, the relative overhead is small (≈ 33%). We found that EMP performs worse than OS primarily due to (1) the overhead of its "real-time circuit optimization" feature, (2) inefficient data buffering when using the network, and (3) virtual function overhead when executing the circuit. OS uses MAGE's runtime, so it does not have these issues. SEAL is faster than OS when the problem fits in memory, but only slightly (less than 20%), indicating that the serialization overhead is not large. When the problem size does not fit in memory, SEAL improves further compared to OS, but remains less than 2× faster than OS.

8.4 Overhead of Swapping Pages

We ran the three scenarios on all 10 workloads, using a 1 GiB memory limit. The results are shown in Fig. 8. We ran 8 trials on different Azure instances (8 different pairs of instances, for garbled circuits) and plot the median; error bars are the quartiles. We additionally ran experiments using a 16 GiB memory limit. We increased the problem sizes so that their memory use exceeded 16 GiB (necessary for the OS scenario) but fit within the 64 GiB available on the virtual machines (necessary for the Unbounded scenario). Our methodology is the same as for the 1 GiB memory limit. We do not include **sort** in our results for the 16 GiB memory limit, because the intermediate bytecodes produced while planning were too large for the local SSD. The results are shown in Fig. 9. MAGE outperforms OS swapping by at least 4× on 7 of the workloads, with improvements of ≈ 12× for **ljoin** and ≈ 10× for **rsum**. Its performance is within 15% of Unbounded for 7 of the workloads (including **sort** from Fig. 8).

MAGE's improvement compared to OS is higher for **binfclayer** and **rmvmul** than for **mvmul**; although all three have similar access patterns, **mvmul** has lower memory intensity because multiplying integers in a garbled circuit has high overhead. For complex access patterns, like **merge** and **sort**, MAGE's improvement is not markedly higher than for simple scans like **ljoin**, **rsum**, and **rstats** (note that both input tables for **ljoin** fit in memory; it is the *output*, populated in order, that does not fit). MAGE is less affected by high memory intensity than OS, allowing it to perform well.

8.5 Overhead of Planning

The time and peak memory use for planning each workload for the MAGE scenario in Fig. 8 and Fig. 9 is shown in Table 1. Note that MAGE's planning is outside of the critical path: for a given circuit, MAGE's planner can be run before the parties' inputs are known. For garbled circuits, although the garbled circuit $\widetilde{C}$ cannot be reused if the computation is re-run, MAGE's memory program *can* be safely reused.

The planning time and final memory program size are linear in the size of the *computation* (size of C), not in the size of the memory demand. Nevertheless, the planning times are generally less than the time to perform the execution and the planner's memory consumption is significantly smaller than the available memory at runtime for all experiments.

Generating memory programs for CKKS is more efficient than for garbled circuits. This is because each instruction for CKKS operates on more memory than for garbled circuits, which means that the problem sizes that fill a given physical memory size tend to require smaller bytecodes for CKKS than for garbled circuits. For example, an instruction operating on integers in a garbled circuit program may operate on a few kilobytes of memory (each bit of each integer is 16 bytes), but for CKKS, each instruction operates on a *vector* of real numbers, whose encrypted size is hundreds of kilobytes.

For CKKS, the final memory programs were < 100 MiB for Fig. 8 and < 1 GiB for Fig. 9. For garbled circuits other than **sort**, they were < 5 GiB for Fig. 8 and < 65 GiB for Fig. 9. For **sort**, it was less than < 25 GiB for Fig. 8. MAGE's planner requires about 4–5× times more storage space than

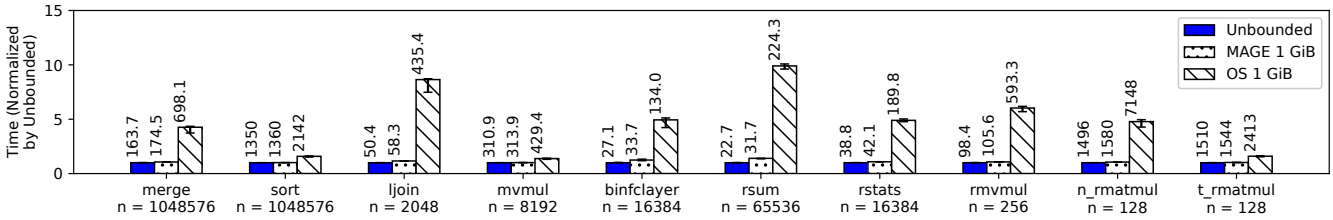


Figure 8: Performance of Unbounded, OS Swapping, and MAGE, normalized by the time for Unbounded; absolute times, in seconds, are printed at the upper left corner of each bar

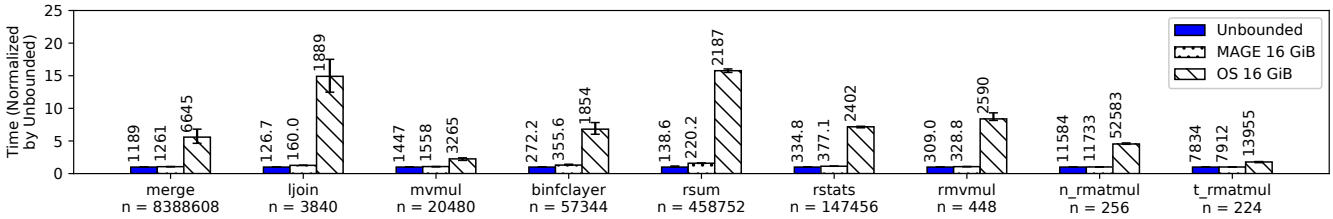


Figure 9: Repeat of Fig. 8, with larger problem sizes and a 16 GiB memory limit (note the larger y-axis scale)

Problem	Time (8)	Mem. (8)	Time (9)	Mem. (9)
merge	38.0	42.6	291.6	299.4
sort	367.3	42.7	N/A	N/A
ljoin	6.7	121.0	23.6	411.4
mvmul	56.0	527.5	298.2	3268
binfclayer	77.2	19.1	1041	165.7
rsum	0.04	9.6	0.29	30.2
rstats	0.04	10.9	0.34	48.5
rmvmul	0.09	16.4	0.24	36.9
n_rmatmul	2.2	246.1	18.6	1927
t_rmatmul	2.3	246.5	12.9	1246

Table 1: Planning times (s) and peak memory use of the planner (MiB) for workloads in Fig. 8 and Fig. 9

the final memory program due to the need to materialize intermediate bytecodes of similar size, but this could be optimized by pipelining stages of MAGE's planner where it is possible to do so (e.g., replacement and scheduling in Fig. 4).

8.6 Impact of Parallelism

We now explore how the relative performance of Unbounded, OS, and MAGE are affected by parallelizing the computation. We did experiments parallelizing the computation across four workers (per party, for garbled circuits). We place each worker on a separate VM instance, each with a separate SSD.

We ran each experiment three times, using the same cluster of machines for all trials, and report the median in Fig. 10. Most experiments follow a similar pattern as Fig. 8, indicating that MAGE's performance gains persist when we parallelize the computation. For two experiments, **merge** and **sort**, MAGE's improvement over OS Swapping visibly increases. Whereas the other workloads are parallelized by splitting the input among the workers in a communication phase at the beginning and then computing independently thereafter, **merge** and **sort** have a communication phase in the *middle* of the computation (several such phases in the case of **sort**). That OS Swapping performs worse for these workloads, but MAGE does not, suggests that the OS virtual memory system might be introducing jitter, which interacts poorly with the communication phase and induces stragglers.

8.7 SMPC in Wide-Area Networks

SC does not always require significant data transfer over the wide area. In HE, computation is done by a single logical party. Even in SMPC, there may be ways for multiple parties to co-locate for an SMPC computation while remaining physically and logically distinct. But in some cases, it is desirable to run SMPC over a wide-area network. We explore this below.

We measure performance of garbled circuits with the two parties hosted on different cloud providers. The garbler was always on Azure in the US West 2 region (Oregon). The evaluator was on Google Cloud (`n2-highcpu-2` [30]). We compare two setups: one where the evaluator was in `us-west1` (Oregon) and one where it was in `us-central1` (Iowa).

Initially, higher latencies and limited single-flow bandwidth limited performance. For example, the round-trip time in the Oregon setup was ≈11 ms, which made OTs a bottleneck.

First, we tuned the local TCP stack, increasing the maximum window size to 32 MiB. Then, we increased the number of OT rounds performed concurrently, pipelining multiple OT rounds over a single connection, which significantly improved performance (Fig. 11a). Additionally, we explore parallelizing the computation, assigning multiple workers to the same machine, so that multiple TCP flows are used. The results are in Fig. 11b. The dashed line at the bottom is the time to run the experiment with both the garbler and evaluator on Azure (taken from Fig. 8). For the Oregon setup, we can come close to the Local performance using two flows. The Iowa setup is more challenging because less bandwidth is available per flow. Using multiple parallel flows helps, but the performance improvement in the Iowa setup is limited by variation in wide-area flow performance, which induces stragglers.

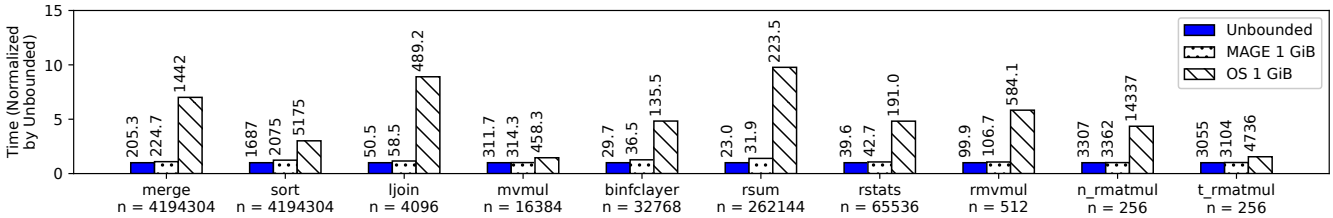
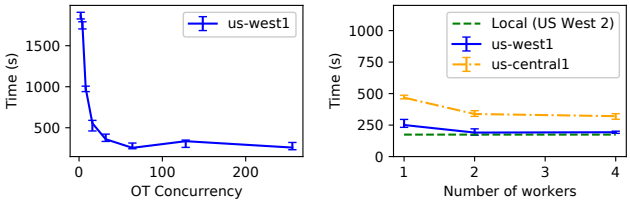


Figure 10: Normalized performance of Unbounded, OS Swapping, and MAGE, parallelized over $p = 4$ workers (per party)



(a) Time to run **merge** vs. number of concurrent OTs

(b) Time to run **merge** vs. number of workers

Figure 11: Wide-area garbled circuit performance in MAGE

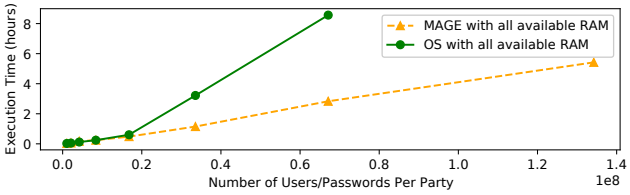


Figure 12: Scaling password reuse detection with MAGE

In both cases, the performance overhead of operating in the wide area is less than the performance overhead of swapping (Fig. 8), indicating that MAGE's techniques confer substantial benefit even in wide-area settings.

8.8 Applications

For these experiments, we did not use `cgroups` to limit RAM. The OS and MAGE setups ran using all of the available RAM.

8.8.1 Detecting Password Reuse

When users reuse a password across multiple websites, they become prone to "credential stuffing" attacks, in which an attacker uses a user's password leaked by one site to compromise that user's account on other sites. To address this problem, sites may wish to identify which of their users reuse their passwords on other sites [81]. Senate [66, Query 2 in §2] proposes a protocol for this. First, the sites arrange to assign user IDs and hash passwords such that they will match *across* sites. Then, they use SMPC to detect which user IDs are shared between the sites and have the same password hash. Note that user IDs and password hashes cannot be shared directly, since they are sensitive (the hashes can be reversed).

We write a two-party version of the password reuse program in MAGE's DSL for garbled circuits, based on Senate's password reuse program. Senate uses a different SMPC protocol, so its results are not directly comparable to ours.

We use MAGE to scale the password reuse program to 2^{27} users per party, which requires 1.125 TiB on each party. A single `D16d_v4` instance does not have enough swap space. Thus, we use four `D16d_v4` instances on Azure for the garbler party, and four `n2-highmem-4` instances on Google Cloud [30] for the evaluator party. As explored in §8.7, we use two workers per instance (total of eight workers per party) to efficiently use wide-area network bandwidth. The results are shown in Fig. 12. For a given time budget, MAGE increases the number of user-password records by $\approx 3\times$. This improvement may have been larger had we been able to obtain `Ddv4`-series instances with a greater swap-space-to-RAM ratio.

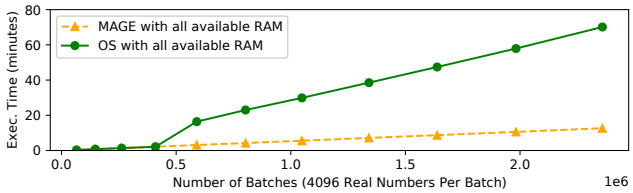


Figure 13: Scaling computational PIR with MAGE

8.8.2 Private Information Retrieval

Private Information Retrieval (PIR) is a family of protocols that allow a user to retrieve a data item at a particular index from a database without the database learning which item was accessed. PIR can be used to support public queries on private data [80]. We evaluate MAGE by using CKKS to instantiate the classic Kushilevitz-Ostrovsky single-server computational PIR scheme [50, §3]. PIR's access pattern is particularly simple—a linear scan over the database—so ad-hoc approaches to prefetching, or multi-threading to improve swap performance, may be quite effective. Our focus is on what MAGE optimizes *automatically*, so we do not include such ad-hoc optimizations in the OS baseline. We use a single worker (thread) to compute the PIR. The database consisted of plaintext data pre-encoded into batches to use with CKKS. We wrote a DSL program that populates the database (with hardcoded elements) and then performs a PIR query on it; the reported measurements are the time to perform the PIR query, not including the time to populate the database. The results are in Fig. 13. For a given time budget, MAGE allows for $\approx 5\times$ as many database elements to be processed.

9 Related Work

Much existing work has looked at high-performance algorithms for SMPC [21, 22, 44, 45, 84] and HE [17, 29]. These works focus on the cryptography, not how to manage a computer's resources to perform large computations efficiently.

A complementary line of work explores tailoring SMPC

computations to a specific application [15,43,68,94]. The goal of MAGE is to perform the same computation more efficiently, so its techniques generalize across different applications. For an application, one may first simplify the computation using application-specific observations, and then execute the resulting computation as efficiently as possible.

Research works including Fairplay [55], HEKM [37], KSS [49], MLB [61], PCF [48], and TinyGarble [73] are frameworks for garbled circuit execution. We described many of them in §2.4. One work [11] explores parallelizing execution of a garbled circuit, using programming language tools to automatically extract parallelism. None of them explore how to efficiently swap memory to storage, as MAGE does.

There already exist many DSLs and compilers for SMPC [34, 36, 51, 60, 82, 89, 93] and HE [13, 23, 78]. These tools often aim to make SC more accessible to non-expert developers, by automatically optimizing the SC program. MAGE addresses the complementary problem of executing the resulting SC circuit more efficiently. To use an existing tool with MAGE (as in Fig. 2), one could modify it to output its optimized circuits in one of MAGE's DSLs, and then run MAGE's planner on that DSL code. Alternatively, one could modify the tool to output a bytecode directly usable by MAGE's planner (e.g., the "Virtual Bytecode" in Fig. 4).

AIFM [70] uses similar C++ language features as MAGE's DSLs. AIFM uses them at runtime for fine-grained memory management. In contrast, MAGE (1) executes DSL programs only to extract the memory access pattern during the planning phase and (2) manages memory at the granularity of pages.

There is an extensive literature concerning memory management in traditional operating systems [3, 4, 24–26]. A related line of work looks at how operating systems can give memory-intensive applications, such as scientific simulations, more control over paging [32]. While these works focus primarily on paging in the classic sense, our work explores memory programming. Additionally, our work, unlike scientific simulations, is capable of *general* computations within SC. Scheduling page movement according to real-time constraints imposed by computation also draws from the real-time scheduling literature [52]. These techniques do not manage memory directly and are complementary to ours.

Some systems in other domains, like neural network training, formulate memory management problems as an integer linear program and use an exponential-time solver [40]. This approach exploits the high-level structure of the application to coarsen the dataflow graph. For MAGE, the dataflow graph is much larger because *general* SC computations do not conform to any particular high-level structure. By operating on a program representation of the circuit (§4.2), MAGE does coarsen the graph, but it nevertheless remains enormous. Thus, we use our staged approach (§6) to find a good approximation.

Some systems use observations of past memory accesses or past working sets (e.g., from prior invocations of a program) to perform targeted prefetching [33, 35, 56, 77, 92] and approximate Belady's algorithm (MIN) [72]. SC's obliviousness and our memory programming approach allow MAGE to compute the memory access pattern without first running the program, and then apply these techniques using the access pattern itself.

The recent DEMAND-MIN [39] algorithm combines MIN with prefetching. DEMAND-MIN tells which item to evict given an access pattern sequence and prefetch sequence fixed in advance. It is not directly applicable to MAGE because MAGE's prefetch sequence is not fixed in advance.

At a technical level, MAGE's planning is similar to register allocation in compiler theory [14, 18, 74, 85]—variables, registers, and memory in register allocation correspond to wire values, slots in memory, and storage swap space in the context of MAGE. The key difference is that register allocators must deal with conditional branches whose outcomes cannot be predicted at compile time. From the perspective of register allocation, the entire circuit that MAGE operates on would be viewed as a single basic block. We discussed a result from register allocation theory for a single basic block in §6.3. Another result is that, for a *fixed* number of registers, there is a linear-time algorithm that can reorder instructions within a structured program to optimize its register allocation [7, §3.2] (though the time is exponential in the number of registers).

10 Conclusion

This paper explores how to efficiently execute SC computations that do not fit in memory. Our key observation is that SC is inherently oblivious. This enables memory programming, in which one computes the access pattern of an SC program in advance and uses it to produce a memory management plan. By using memory programming to preplan data transfers between memory and storage, MAGE runs SC up to an order of magnitude faster than the OS virtual memory system and can execute some SC programs at nearly in-memory speeds.

Some non-SC programs, like plaintext neural network inference and programs designed for hardware enclaves like Intel SGX, are also oblivious. Applying memory programming to such workloads is an interesting direction for future work.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Nadav Amit, for their helpful feedback. We would also like to thank Katerina Sotiraki and other students/postdocs from the RISELab Security Group for their feedback on early drafts.

This work is supported by NSF CISE Expeditions Award CCF-1730628, NSF CAREER 1943347, and gifts from the Sloan Foundation, Bakar Fellows Program, Alibaba, Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. This research is also supported in part by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1752814. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] J. Bater, G. Elliott, V. Eggen, S. Goel, A. Kho, and J. Rogers. SMCQL: Secure querying for federated databases. *VLDB*, 10(6), 2017.

[2] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC*. ACM, 1990.

[3] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Syst. J.*, 5(2), 1966.

[4] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *CACM*, 12(6), 1969.

[5] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *S&P*. IEEE, 2013.

[6] O. Biçer. Efficiency optimizations on Yao's garbled circuits and their practical applications. Master's thesis, Istanbul Şehir University, 2017. Chapters 3 and 4.

[7] H. Bodlaender, J. Gustedt, and J. A. Telle. Linear-time register allocation for a fixed number of registers. In *SODA*. SIAM, 1998.

[8] J. Bonwick. The slab allocator: An object-caching kernel. In *USENIX Summer Technical Conference*. USENIX Association, 1994.

[9] D. P. Bovet and M. Cesati. Page frame reclaiming. In *Understanding the Linux Kernel*, chapter 17, page 679. O'Reilly Media, 2006.

[10] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. Cryptology ePrint Archive, Report 2019/448, 2019. https://eprint.iacr.org/2019/448.

[11] N. Buescher and S. Katzenbeisser. Faster secure computation through automatic parallelization. In *USENIX Security*. USENIX, 2015.

[12] Cape Privacy. https://medium.com/dropoutlabs.

[13] S. Carpov, P. Dubrulle, and R. Sirdey. Armadillo: A compilation chain for privacy preserving applications. In *SCC*. ACM, 2015.

[14] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1), 1981.

[15] H. Chen, M. Kim, I. P. Razensteyn, D. Rotaru, Y. Song, and S. Wagh. Maliciously secure matrix multiplication with applications to private deep learning. 2020. https://eprint.iacr.org/2020/451.

[16] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*. Springer, Cham, 2017.

[17] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *ASIACRYPT*. Springer, Berlin, Heidelberg, 2016.

[18] K. D. Cooper and L. T. Simpson. Live range splitting in a graph coloring register allocator. In *CC*. Springer, Berlin, Heidelberg, 1998.

[19] Curv. Curv | digital asset security infrastructure. https://www.curv.co/.

[20] D. McDaniel. Virtual machines best practices: Single VMs, temporary storage and uploaded disks. https://azure.microsoft.com/en-us/blog/virtual-machines-best-practices-single-vms-temporary-storage-and-uploaded-disks/, 2014.

[21] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority – or: breaking the SPDZ limits. In *ESORICS*. Springer, Berlin, Heidelberg, 2013.

[22] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011. https://eprint.iacr.org/2011/535.

[23] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi. EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. ACM, 2020.

[24] P. J. Denning. Thrashing: its causes and prevention. In *AFIPS*. ACM, 1968.

[25] P. J. Denning. Virtual memory. *CSUR*, 2(3), 1970.

[26] P. J. Denning. Working sets past and present. *IEEE Trans. Softw. Eng.*, SE-6(1), 1980.

[27] Duality. https://dualitytech.com/.

[28] M. Farach and V. Liberatore. On local register allocation. In *SODA*. SIAM, 1998.

[29] C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*. Springer, Berlin, Heidelberg, 2012.

[30] Google Cloud. Machine types. https://cloud.google.com/compute/docs/machine-types.

[31] C. Guo, J. Katz, X. Wang, C. Weng, and Y. Yu. Better concrete security for half-gates garbling (in the multi-instance setting). Cryptology ePrint Archive, Report 2019/1168, 2019. https://eprint.iacr.org/2019/1168.

[32] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *ASPLOS*. ACM, 1992.

[33] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan. Learning memory access patterns. In *ICML*, 2018.

[34] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *S&P*. IEEE, 2019.

[35] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun. I/O acceleration with pattern detection. In *HPDC*. ACM, 2015.

[36] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ANSI C. In *CCS*. ACM, 2012.

[37] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. USENIX, 2011.

[38] Inpher. https://inpher.io/.

[39] A. Jain and C. Lin. Rethinking belady's algorithm to accommodate prefetching. In *ISCA*. ACM/IEEE, 2018.

[40] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, K. Keutzer, I. Stoica, and J. Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *MLSys*, 2020.

[41] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. IEEE, 2008.

[42] X. Jiang, M. Kim, K. Lauter, and Y. Song. Secure outsourced matrix computation and application to neural networks. ACM, 2018.

[43] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security*. USENIX, 2018.

[44] M. Keller, E. Orsini, and P. Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *CCS*. ACM, 2016.

[45] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*. Springer, Cham, 2018.

[46] Keyless. Keyless | zero-trust passwordless authentication. https://keyless.io/.

[47] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*. Springer, Berlin, Heidelberg, 2008.

[48] B. Kreuter, B. Mood, A. Shelat, and K. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *USENIX Security*. USENIX, 2013.

[49] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*. USENIX, 2012.

[50] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single databse, computationally-private information retrieval. In *FOCS*. IEEE, 1997.

[51] C. Liu, X. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *S&P*. IEEE, 2015.

[52] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1), 1973.

[53] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *CCS*. ACM, 2017.

[54] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*. ACM, 2005.

[55] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *USENIX Security*. USENIX, 2004.

[56] H. Al Maruf and M. Chowdhury. Effectively prefetching remote memory with leap. In *ATC*. USENIX, 2020.

[57] Microsoft Azure. Ddv4 and Ddsv4-series. https://docs.microsoft.com/en-us/azure/virtual-machines/ddv4-ddsv4-series, 2020.

[58] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. Delphi: A cryptographic inference service for neural networks. In *USENIX Security*. USENIX, 2020.

[59] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *S&P*. IEEE, 2017.

[60] B. Mood, D. Gupta, H. Carter, K. R. B. Butler, and P. Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *EuroS&P*. IEEE, 2016.

[61] B. Mood, L. Letaw, and K. Butler. Memory-efficient garbled circuit generation for mobile devices. In *FC*. Springer, Berlin, Heidelberg, 2012.

[62] J. Nielsen. Nielsen's law of Internet bandwidth. Accessed: May 26, 2020.

[63] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *S&P*. IEEE, 2013.

[64] T. Peng. Shared machine learning: Ant financial's solution for data privacy. https://medium.com/syncedreview/shared-machine-learning-ant-financials-solution-for-data-privacy-8069cffe7bb6.

[65] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *ASIACRYPT*. Springer, Berlin, Heidelberg, 2009.

[66] R. Poddar, S. Kalra, A. Yanai, R. Deng, R. A. Popa, and J. M. Hellerstein. Senate: A maliciously-secure MPC platform for collaborative analytics. In *USENIX Security*. USENIX, 2021.

[67] B. Randell. A note on storage fragmentation and program segmentation. *CACM*, 12(7), 1969.

[68] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In *USENIX Security*. USENIX, 2019.

[69] M. Rosulek. A brief history of practical garbled circuit optimizations, 2015. https://simons.berkeley.edu/talks/mike-rosulek-2015-06-09, https://www.youtube.com/watch?v=FTxh908u9y8.

[70] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *OSDI*. USENIX, 2020.

[71] Microsoft SEAL (release 3.6). https://github.com/Microsoft/SEAL, 2020. Microsoft Research, Redmond, WA.

[72] Z. Song, D. S. Berger, K. Li, and W. Lloyd. Learning relaxed Belady for content distribution network caching. In *NSDI*. USENIX, 2020.

[73] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *S&P*. IEEE, 2015.

[74] O. Traub, G. Holloway, and M. D. Smith. Quality and speed in linear-scan register allocation. In *SIGPLAN*. ACM, 1998.

[75] Unbound. https://www.unboundtech.com/.

[76] Laakeri (https://cs.stackexchange.com/users/95646/laakeri). Is there an algorithm to minimize working set during a topological traversal? Computer Science Stack Exchange, 2020. https://cs.stackexchange.com/q/120274.

[77] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *ASPLOS*. ACM, 2021.

[78] A. Viand, P. Jattke, and A. Hithnawi. SoK: Fully homomorphic encryption compilers. In *S&P*. IEEE, 2021.

[79] N. Volgushev, M. Schwarzkopf, B. Getchell, M. Varia, A. Lapets, and A. Bestavros. Conclave: secure multiparty computation on big data. In *EuroSys*. ACM, 2019.

[80] F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia. Splinter: Practical private queries on public data. In *NSDI*. USENIX, 2017.

[81] K. C. Wang and M. K. Reiter. How to end password reuse on the web. In *NDSS*. Internet Society, 2019.

[82] X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[83] X. Wang, S. Ranellucci, and J. Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS*. ACM, 2017.

[84] X. Wang, S. Ranellucci, and J. Katz. Global-scale secure multiparty computation. In *CCS*. ACM, 2017.

[85] C. Wimmer and H. Mössenböck. Optimized interval splitting in a linear scan register allocator. In *VEE*. ACM, 2005.

[86] S. Yakoubov. A gentle introduction to Yao's garbled circuits, 2017. http://web.mit.edu/sonka89/www/papers/2017ygc.pdf.

[87] A. C.-C. Yao. Protocols for secure computations. In *FOCS*. IEEE, 1982.

[88] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS*. IEEE, 1986.

[89] S. Zahur and D. Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. https://eprint.iacr.org/2015/1153.

[90] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*. Springer, Berlin, Heidelberg, 2015.

[91] Zcash. Parameter generation. https://z.cash/technology/paramgen/.

[92] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr. Fast restore of checkpointed memory using working set estimation. In *VEE*. ACM, 2011.

[93] W. Zheng, R. Deng, W. Chen, R. A. Popa, A. Panda, and I. Stoica. Cerebro: A platform for multi-party cryptographic collaborative learning. In *USENIX Security*. USENIX, 2021.

[94] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *S&P*. IEEE, 2019.

[95] R. Zhu, D. Cassel, A. Sabry, and Y. Huang. NANOPI: Extreme-scale actively-secure multi-party computation. In *CCS*. ACM, 2018.

A Artifact Appendix

Abstract

Our artifact consists of a MAGE prototype and scripts to use it to run our experiments from §8. The MAGE prototype can execute SC efficiently even when the computation does not fit in memory. It does so by using memory programming to provide a very efficient virtual memory abstraction. Our prototype supports distributing an SC computation across workers that communicate over the network, allowing for parallel and distributed SC execution. The MAGE prototype presently supports two SC protocols: garbled circuits and CKKS. It follows the layered architecture described in §4.3.

Scope

Our artifact can be used to validate our central claim that, using memory programming, MAGE can execute SC computations that do not fit in memory at nearly in-memory speeds. Specifically, our artifact can be used to validate the performance claims made in the figures and table in §8. Our submitted artifact package allowed the artifact evaluation committee to reproduce those results present in our submitted paper; we have since added support for reproducing the measurements we have added since the original submission.

Our artifact can also be used to run SC computations unrelated to our evaluation of MAGE in §8. The user can describe a custom SC computation using a DSL internal to C++, and then use our MAGE prototype to generate a memory program for it and execute it efficiently.

Contents

Our artifact comprises (1) a prototype of MAGE and (2) scripts to run experiments from §8.

Prototype. Our MAGE prototype includes:

- The planner and interpreter for the MAGE system.
- A utility program to read the bytecode format used by our implementation and print a memory program in human-readable form.
- Implementations of the workloads used in our evaluation (§8.1) in MAGE's DSLs.
- Utility programs to prepare inputs for these workloads.
- A wiki page that walks the user through using our MAGE prototype to perform a computation.

Scripts. Our scripts to run our experiments include:

- A program, `magebench.py`, that can spawn cloud instances on Microsoft Azure and Google Cloud and run experiments on the resulting cloud setup. The command line parameters passed to this program can be used to specify the cloud setup and experiments to run; the user can change these command line parameters to change aspects of the setup (e.g., number of workers, memory size, problem size, etc.).
- A README file that describes how to use `magebench.py` to run our experiments from §8 and obtain log files containing the results.
- An IPython notebook to produce graphs from the log files output by `magebench.py`.
- Utility scripts to help automate invoking `magebench.py` to run experiments from §8.

Hosting

Our artifact is available on GitHub. Our MAGE prototype is available at https://github.com/ucbrise/mage and our scripts to run our experiments are available at https://github.com/ucbrise/mage-scripts. The version we provided to the artifact evaluation committee is marked in both repositories using the `osdi21ae` tag. However, we encourage users to use the latest versions of each repository (on the `main` branch), as they include the newest features and bug fixes, including scripts for additional experiments in §8.

Requirements

We developed and tested our artifact on Intel x86-64 systems running Ubuntu 20.04. We used `clang++` 10.0.0 to compile our MAGE prototype. The `magebench.py` script spawns cloud instances with an environment appropriate for building and running our MAGE prototype. Spawning those cloud instances requires a subscription to Microsoft Azure and Google Cloud. The particular software dependencies for our artifact are specified in the README files of our two GitHub repositories.

Workflow

To use our MAGE prototype, the user first writes a configuration file in YAML describing the execution setup (e.g., network information and swap file for each worker, number

of concurrent OTs for garbled circuits, etc.). For SMPC, information needed only by other parties (e.g., the swap file for other parties' workers) can be omitted from the configuration file. Next, the user writes a program in a DSL internal to C++ specifying the computation to run. Then, the user runs MAGE's planner, which accepts the DSL program and configuration file as input, for each worker the user will run, and outputs a file containing a memory program for each worker. The user prepares a file for each worker describing that worker's input for the computation. Finally, the user runs MAGE's interpreter for each worker, which accepts files containing the memory program, configuration, and input data and writes a file containing the program's output. Further details are given in the README file and wiki pages of the `mage` repository on GitHub.

To use our script to run experiments, the user invokes `magebench.py` to spawn cloud virtual machines. The user can then invoke `magebench.py` to run MAGE on those cloud virtual machines, copy the resulting log files to the machine where `magebench.py` is run, and finally, deallocate the cloud virtual machines. Further details are given in the README file of the `mage-scripts` repository on GitHub.