Towards Generic MPC Compilers via Variable Instruction Set Architectures (VISAs)

Yibin Yang
Georgia Institute of Technology, USA
yyang811@gatech.edu

Stanislav Peceny
Georgia Institute of Technology, USA
stan.peceny@gatech.edu

David Heath
University of Illinois Urbana-Champaign, USA
daheath@illinois.edu

Vladimir Kolesnikov
Georgia Institute of Technology, USA
kolesnikov@gatech.edu

ABSTRACT

In MPC, we usually represent programs as circuits. This is a poor fit for programs that use complex control flow, as it is costly to compile control flow to circuits. This motivated prior work to emulate CPUs inside MPC. Emulated CPUs can run complex programs, but they introduce high overhead due to the need to evaluate not just the program, but also the machinery of the CPU, including fetching, decoding, and executing instructions, accessing RAM, etc.

Thus, both circuits and CPU emulation seem a poor fit for general MPC. The former cannot scale to arbitrary programs; the latter incurs high per-operation overhead.

We propose *variable instruction set architectures* (VISAs), an approach that inherits the best features of both circuits and CPU emulation. Unlike a CPU, a VISA machine repeatedly executes entire program *fragments*, not individual instructions. By considering larger building blocks, we avoid most of the machinery associated with CPU emulation: we directly handle each fragment as a circuit.

We instantiated a VISA machine via garbled circuits (GC), yielding constant-round 2PC for arbitrary assembly programs. We use improved branching (Stacked Garbling, Heath and Kolesnikov, Crypto 2020) and recent Garbled RAM (GRAM) (Heath et al., Eurocrypt 2022). Composing these securely and efficiently is intricate, and is one of our main contributions.

We implemented our approach and ran it on common programs, including Dijkstra's and Knuth-Morris-Pratt. Our 2PC VISA machine executes assembly instructions at 300Hz to 4000Hz, depending on the target program. We significantly outperform the state-of-the-art CPU-based approach (Wang et al., ESORICS 2016, whose tool we re-benchmarked on our setup). We run in constant rounds, use 6× less bandwidth, and run more than 40× faster on a low-latency network. With 50ms (resp. 100ms) latency, we are 898× (resp. 1585×) faster on the same setup.

While our focus is MPC, the VISA model also benefits CPU-emulation-based Zero-Knowledge proof compilers, such as ZEE and EZEE (Heath et al., Oakland'21 and Yang et al., EuroS&P'22).

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3616664

CCS CONCEPTS

• **Security and privacy → Cryptography**; • **Theory of computation → Cryptographic protocols**.

KEYWORDS

MPC; General Purpose Programs; Garbled Circuits

ACM Reference Format:
Yibin Yang, Stanislav Peceny, David Heath, and Vladimir Kolesnikov. 2023. Towards Generic MPC Compilers via Variable Instruction Set Architectures (VISAs) . In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3616664

1 INTRODUCTION

Secure multi-party computation (MPC) allows mutually untrusting parties to execute programs on their private inputs while revealing only the output. MPC has become relevant in academia and industry. It has been commercially deployed in online auctions, electronic voting, financial technology, and has found many use cases in medicine, privacy-preserving machine learning, and distributed databases.

Typically in MPC, we encode programs as circuits. While any bounded program can be compiled to a circuit, the compiled circuit is often *much* larger than the source program. Real world programs (1) access large arrays of data and (2) use complex control flow. Compiling these two program features often results in huge circuits, and MPC cost scales with the size of the circuit. If we wish to enable secure computation of real-world programs, we must circumvent the cost imposed by compiling these features to circuits.

While the issue of array access can be resolved via oblivious RAM (ORAM) [GO96] or garbled RAM (GRAM) [LO13], complex control flow has gone largely unaddressed.

Straight-line execution. Indeed, most existing MPC tools "solve" the control flow problem by *disallowing* complex control flow. Most existing MPC toolchains require that the programmer hand-annotate each loop with a hard-coded upper bound on the number of loop iterations [HHNZ19]. With these annotations, the program becomes a simple *straight-line* program, compatible with the circuit model. A compiler can now unroll each loop precisely the specified number of times, then compile each iteration into gates.

This approach is problematic. At best, annotating programs is an annoyance. At worst, hard-coded loop bounds *ruin performance*,

since we must execute each loop iteration, even if the loop should terminate early. Many programs are awkward to write and will have wildly wrong asymptotic complexity. In other words, the programmer is left with an incomplete programming environment where she cannot write every program she might need.

CPU emulation. There is another approach that addresses the control flow problem. Instead of evaluating the program directly, use MPC to *emulate a CPU*, then run the program on that CPU. To achieve this, we must fully emulate a CPU, including its program counter, register file, ALU, and main memory. At each step, the CPU will look up and decode the next instruction, load/store registers, run arithmetic operations, and read/write main memory. In this way, the parties can securely evaluate one instruction at a time.

CPU emulation can securely evaluate arbitrary programs, but at a cost. When we emulate a CPU, we pay to evaluate not just the program, but also the machinery inside the CPU. In comparison to straight-line execution, CPU emulation incurs *very* high per-operation cost. In straight-line execution, the arguments to each program operation are decided statically; in a CPU, the arguments are *dynamic* and must be moved into and out of the register file. Similarly, in straight-line execution, the operation to be performed at each step is decided statically; in a CPU, we must look up the next instruction from a large memory, then conditionally dispatch over each operation in the ISA. In short, while CPU emulation solves the control flow problem, it introduces high overhead.

Full version. Full version of this paper is available at [YPHK23].

1.1 Case study: Dijkstra's algorithm

We illustrate the challenge of handling general programs in MPC. Consider Dijkstra's algorithm[1] (Figure 1).

Dijkstra's is a graph algorithm that computes the shortest path between a source node (`b[0]`) and a target node (`b[1]`). Assume that the graph, the source, and the target are private inputs (e.g., secret-shared between the parties). Both straight-line execution and CPU emulation struggle with this small program.

Straight-line execution. Straight-line execution-based tools will generally achieve the wrong cost for the algorithm. Consider a graph (V, E). Even if we suppose that $|V|$ and $|E|$ are public, this simple program presents a challenge to straight-line execution. The problem is that the loop on lines 26–31 dynamically accesses each edge from a graph node. The number of edges from this node is private, so the loop must be unrolled $|E|$ times to accommodate the maximum possible number of edges. In a cleartext execution of Dijkstra's, this inner loop will in total iterate only $O(|E|)$ times; in this unrolled version, the inner loop will iterate $O(|E| \cdot |V|)$ times.

Even worse, suppose that Dijkstra's is not the full program, but rather is a subprocedure of a larger program. Here, $|V|$ and $|E|$ are likely dynamic and should be kept secret. In this case, straight-line execution-based tools must pessimistically assume that $|V|$ and $|E|$ are maximal, ruining performance.

[1]For performance, Dijkstra's algorithm may be implemented with a *priority queue* containing partial solutions sorted by distance from the start node. Standard Dijkstra is based on a simple array, as is also done in [WGMK16]. We use standard Dijkstra for illustration and direct performance comparison with [WGMK16].

```
1   #define MAX 100
2   #define MAX_INT 1000000
3   int dijkstra(int *a, int *b) {
4       int n = a[0];
5       int e = a[1];
6       int * node = a + 2;
7       int * edge = a + 2 + 101;
8       int * weight = a + 2 + 401;
9       int vis[MAX];
10      int dis[MAX];
11      dis[b[0]] = 0;
12      int i = 0;
13      while (i < n) {
14          int bestj = -1, bestdis = MAX_INT;
15          int j = 0;
16          while (j < n) {
17              if( vis[j] == 0 && dis[j] < bestdis ) {
18                  bestj = j;
19                  bestdis = dis[j];
20              }
21              j++;
22          }
23          vis[bestj] = 1;
24          j = node[bestj];
25          int bound = node[bestj+1];
26          while (j < bound) {
27              int newDis = bestdis + weight[j];
28              if(newDis < dis[edge[j]])
29                  dis[edge[j]] = newDis;
30              j++;
31          }
32          i++;
33      }
34      return dis[b[1]];
35  }
```

Figure 1: Dijkstra's algorithm written in C. Each vertical line on the left denotes a contiguous string of instructions that are grouped into a fragment. I.e., this program has seven fragments.

ObliVM [LWN+15] showed that for Dijkstra's algorithm and if $|V|$ and $|E|$ are public, the straight-line approach can reclaim the loop asymptotics via *loop coalescing*. Using loop coalescing, we can flatten the nested loop on lines 13–33 into a single loop with an internal conditional. Then, the number of iterations of this top level loop is a function of $|V|$ and $|E|$, so it is possible to properly bound the loop. See further discussion in Section 3.

While loop coalescing can solve this particular problem, it places a significant burden on the programmer: the programmer must now reason about and properly specify upper bounds on *coalesced* loops. This may be expensive if $|V|$ and $|E|$ are secret, such as if Dijkstra's is nested inside another data-dependent loop, requiring costly further coalescing or excessive padding. This syntactic transformation produces expensive code that is difficult to further optimize.

CPU emulation. CPU emulation correctly implements Dijkstra's asymptotics[2], but incurs significant concrete cost.

The state-of-the-art CPU emulator implements a sufficient subset of the MIPS instruction set [WGMK16] to handle Dijkstra's. This CPU stores the compiled assembly program, the register file, and the main memory in three separate RAMs. [WGMK16] implements RAM using either Circuit ORAM [WCS15] or trivial linear scans, depending on the size of the needed array. Their CPU proceeds by continually fetching and executing instructions.

Storing the program in RAM and applying the fetch-and-execute paradigm discards all useful *static information*, some of which [WGMK16] *manually* reclaims by implementing various heuristics, such as periodic (rather than per-instruction) RAM access. Even applying this heuristic, their number of main memory accesses is suboptimal. Further, they must always access smaller memories to fetch instructions and to read/write registers. Their ALU decodes the instruction and conditionally executes the operation for each instruction type that is statically possible at a given step. As a result, each CPU step is a large circuit that often improves on the circuit-based computation only for problem instances where MPC is impractical.

Our approach, discussed next, systematically optimizes away many of the principal inefficiencies of [WGMK16] and results in significantly improved performance. For instance, for Dijkstra's with 100 nodes and 300 edges and when run on the *same* setup, our VISA machine uses 5.8× fewer RAM accesses, consumes 7.3× less bandwidth, and runs 44.9× faster. We are 1585× faster on a 100ms-latency network.

Our solution: VISA machines. The state of the art presents a dichotomy: CPU emulation or straight-line programs.

In this work, we suggest and explore a hybrid approach to handling arbitrary programs inside MPC. Our *variable instruction set architecture machine*, or VISA machine, handles programs with arbitrary control flow, but avoids most of the overhead of the CPU emulation approach. It uses the statically available context to optimize the scope (and hence the cost) of each execution step.

In short, a VISA machine is distinct from a CPU in that it does not repeatedly execute instructions, but rather repeatedly executes entire *fragments* of the source program. Each fragment is an arbitrarily long straight-line portion of the source program text. The basic advantage of this is that we can cheaply handle each fragment as a circuit. While we still need CPU-like machinery to coordinate the execution of the fragments and ensure privacy, the amount of needed machinery is substantially reduced.

1.2 Contribution

We propose *variable instruction set architectures*, a basic approach to evaluating arbitrary programs inside MPC. We believe that VISAs are *the* sensible approach to executing arbitrary programs in MPC. VISAs do not limit the programmer to straight-line programs, and they do not incur the high overhead of a basic CPU. A VISA adapts to the target program of interest, an appropriate choice for MPC where we generally assume that the parties agree on a program.

In more detail, we:

- Introduce and motivate the VISA model.
- Construct a complete VISA-based secure two-party computation (2PC) toolchain for assembly programs. Our toolchain is implemented via garbled circuits (GC).
- Resolve technical issues needed to combine core components of a GC-based VISA machine: GC conditional branching [HK20, HK21b] and Garbled RAM [HKO21].
- Formalize our instantiation as a *garbling scheme* [BHR12] and prove the resulting formalism secure. Our garbling scheme securely evaluates arbitrary assembly programs written in our ISA. Using garbling schemes as the underlying mechanism has two key benefits.
  - First, we dramatically decrease the number of communication rounds, resulting in orders of magnitude improvement (see Section 7.4.3). Prior work [WGMK16, Kel17] used tens of rounds *per CPU step*, while we require one message plus an OT for the entire 2PC.
  - Second, our technique can be elevated to the covert, PVC, and malicious models using standard techniques.
- We implemented VISA machine including, significantly, the first implementation of Garbled RAM [HKO21].
- Experimentally evaluate performance of our toolchain. We ran our VISA machine on a number of assembly benchmarks, including Dijkstra's, Knuth-Morris-Pratt, and a private set intersection benchmark from [WGMK16]. Our results indicate significant improvement over the prior best approach to arbitrary assembly programs [WGMK16]: we run in constant rounds, use 4–7× less bandwidth, use 5–10× fewer RAM accesses, and run 40–70× faster (up to 1585× with 100ms latency), yielding a machine that executes assembly instructions at 300–4000Hz. We also experimentally show our work, as expected, overtakes circuit-based 2PC (EMP [WMK16]) even for small programs with non-trivial control flow.
- We plan to open source and maintain a cleaned version of our prototype toolchain.
- While our focus is on MPC, the VISA model also directly applies to CPU-emulation-based Zero-Knowledge Proof (ZKP) compilers, such as ZEE and EZEE [HYDK21, YHKD22]. Indeed, they face similar problems of more efficient CPU design (e.g., fragmentation and stacking), ZK ORAM integration with branching, etc., and the VISA approach is similarly beneficial to ZKP compiler work. We leave specific instantiations of ZKP VISA as exciting future work.

Recent breakthrough GC and MPC improvements on free branching [HK20, HK21b, HKP20, HKP21] and efficient GRAM [HKO21] removed fundamental technical roadblocks needed to move away from straight-line circuit execution. We believe that our hybrid approach – contextual fragment-based execution engines – will underlie the next generation of 2PC and MPC toolchains. This paper initiates this direction and sets the stage for future cryptographic and interdisciplinary work that will likely involve programming language, static analysis, and compiler techniques, and that will interface with high-level programming languages.

[2]To be pedantic, the CPU emulation approach achieves the correct asymptotics modulo polylog factors imposed by ORAM/GRAM. Neither CPU emulation nor straight-line execution, nor indeed our approach, can avoid polylog overhead from ORAM/-GRAM [LN18].

2 OVERVIEW

We at a high level introduce our model and explain the fundamental benefits of our approach. We then introduce lower-level technical challenges and briefly outline our approach to solving them.

Our basic observation is that CPU emulation is a blunt generic mechanism: CPUs in cleartext machines are static devices that can execute *each* step of *any* program. But in MPC, the program is public, and there is no need to use a fixed generic set of instructions. Instead, we can derive our machine's 'instruction types', which we call *fragments*, from the target program itself.

Each fragment can be arbitrarily large and complex, so long as it does not contain data-dependent loops. We can generate custom circuitry tailored to each fragment, avoiding the need to mechanistically execute the fragment one instruction at a time. Thus, once our machine enters a fragment, we pay essentially no overhead to execute that fragment. In this sense, we obtain the benefit of straight-line execution.

At the same time, our machine dynamically dispatches over the fragments, so we can handle all possible execution paths. In this sense, we obtain the benefit of CPU emulation.

Our execution engine does not necessarily need to dynamically dispatch over each program fragment at each step. At each step it is sufficient to only guarantee execution of fragments that may occur at *this* step. In many useful programs, this *active set* is *much* smaller and consists of cheaper fragments than the full set.

Program fragments are generated by a compiler. There are many choices for how to fragment a program, and good fragmentation is crucial to performance. We discuss related trade-offs (see Section 5.4).

2.1 Notation

Our execution engine repeatedly conditionally dispatches over varying sets of fragments chosen from the target program. We call the specification of a machine that operates this way a *variable instruction set architecture* (VISA). A *VISA machine* instantiates a VISA specification. Our VISA machine, which we call GAR, is implemented via GC; of course, one could implement a VISA machine from different primitives, such as a secret-sharing-based protocol and off-the-shelf ORAM.

At each step i, a VISA machine can execute any fragment in the *active set* of step i. We compose each fragment from many *base instructions* in the program text. Note we thus consider two kinds of instructions: *base* instructions are typical low-level assembly instructions, whereas *fragments* are the instructions of a VISA and are composed from multiple base instructions. Fragments are automatically chosen by a type of compiler that we call a *fragmentation strategy*; our GAR construction includes a built-in fragmentation strategy.

In the remainder of this section, we explain and motivate VISA machines in more detail. We explain our advantages by referring to Dijkstra's algorithm (Figure 1).

2.2 VISA Advantages

VISA machines do not repeatedly execute instructions, but rather repeatedly execute entire *fragments* of the source program. This leads to several important advantages:

Free register file. As each fragment is a straight-line piece of code, we do not need to dynamically store and access local variables from a register file. Instead, like the straight-line approach, a VISA machine routes arguments to operations directly and without cryptographic cost.

We still pay to route the content of the register file *between* fragments, but *within* a single fragment, the register file is free.

Example 2.1. Consider line 18 of Dijkstra's (Figure 1). Under CPU emulation, this simple assignment requires reading `j` from and writing `bestj` to the register file. In practice, these would be implemented by linear scans of a modest array. Linear scans are expensive. As a reference point, suppose the register file holds 16 32-bit registers. Using state-of-the-art GC, *each* linear scan of this register file costs $\approx$ 16KB of communication. In the CPU emulation approach, this cost is paid multiple times per CPU cycle. In our VISA machine, this overhead is erased: to handle line 18 the parties may simply agree to name certain wires in the fragment circuit `bestj`.

No instruction memory. Programs execute fewer fragments than they do base instructions. Thus, when the VISA machine dynamically decides which fragment to execute next, the space of choices is smaller. This means that the VISA machine does not need to store fragments in an instruction memory. Instead, we conditionally dispatch over an integer that indicates which of the small number of statically known fragments should be executed next. This eliminates many usages of ORAM/GRAM.

Example 2.2. In our ISA, Dijkstra's has 56 instructions[3] but only 7 fragments. (Our actual fragmentation is more nuanced; see Section 5.4.) At each step, we conditionally execute only those fragments that are possible. As a simple example, on the first cycle of Dijkstra's, our VISA machine unconditionally executes the fragment on lines 4–12, since this is statically the only fragment possible. We track the fragments that are possible at each step by tracing the target program's control flow graph.

Fewer conditional choices. Each fragment implements a larger portion of the overall execution than does each instruction. This is significant because there is overhead associated with conditionally executing code inside MPC, whether classically or by stacking [HK20]. Since we execute fewer fragments than CPU emulation executes instructions, we make fewer conditional decisions, and hence pay the overhead of conditional branching fewer times. With SGC, this advantage manifests in the fact that we need fewer SGC multiplexer gadgets [HK20, HK21b]. Importantly, for small branches, these gadgets dominate the cost of SGC.

Example 2.3. Running Dijkstra's with $|V| = 100$ and $|E| = 300$ involves executing $198,814$ instructions, and hence making $198,814$ conditional decisions. In contrast, we need only execute $21,800$ fragments, and hence make only $21,800$ conditional decisions.

Fewer data RAM accesses. Since each fragment is static, we know precisely how many times each fragment must move data to/from main memory. This allows a VISA machine to access memory less

[3] For readability, Figure 1 is written in C; our machine manipulates low level assembly, and each line of C code can correspond to multiple assembly instructions. We include assembly code for Dijkstra's and for our benchmark programs in the full version.

often than a CPU, since in a CPU it is possible that each instruction is a memory access.

Example 2.4. Consider again line 18 of Dijkstra's. Under CPU emulation, the CPU cannot statically deduce that the current instruction is *not* a RAM access, so when emulating line 18, it must perform a RAM access. Our VISA machine eliminates this access.

The sum advantage of our approach as compared to CPU emulation is well illustrated by again considering line 18 of Dijkstra's. Under CPU emulation, this instruction will involve fetching and decoding the instruction, linearly scanning the register file multiple times, conditionally executing the various instruction types, and accessing main memory. Each of these actions are expensive. In our VISA-based approach, line 18 is free of cryptographic cost.

2.3 VISA Technical Challenges and Solutions

Our core contribution is the introduction of VISA-based MPC. Efficiently implementing an MPC VISA machine presents crypto- and system-technical challenges; we discuss the main challenges here.

Managing the active set. Inside a fragment, we have full static knowledge of the straight-line code, so we can directly and efficiently compile the code to a circuit. However, a VISA machine must conditionally execute fragments in the active set at each step.

The cost of this conditional dispatch is greatly improved thanks to the recent line of work on MPC conditional branching, in particular Stacked Garbling (SGC) [HK20, HK21b]. By integrating SGC, we can conditionally dispatch over active set fragments with communication proportional to a single (largest) fragment. Although SGC improves communication, it still requires computation: for b fragments, the computational cost scales with $O(b \log b)$ [HK21b]. Thus, we must not allow the active set to grow too large.

Further, SGC-based conditional branching incurs communication cost that scales with the size of the conditional's *interface*, i.e., the number of input/output wires, with additional factor dependent on the number of branches b. This cost imposes constraints on the efficiency of using small fragments, and impacts the utility of breaking down fragments, e.g., in alignment with RAM accesses.

In this work, we do not significantly optimize fragments, leaving it as crucial and significant future work. Our fragments are syntactically derived from the control flow structure of the assembly program. This choice is sufficient for modest programs. We envision that future work can use compiler techniques and static analysis to more intelligently select fragments. For example, a fragment can be split into pieces, or multiple fragments can be combined into one. We emphasize the complexity of this problem space: a good solution should simultaneously consider the size of each active set, the size of fragments, the number of RAM accesses, the per-fragment overhead, such as the size of the interface to SGC, etc.

Stacked Garbling with RAM access. Using SGC to conditionally evaluate fragments introduces a subtle technical challenge in handling RAM accesses within fragments. For multiple technical reasons, it is not possible or desired to access RAM directly from inside an SGC conditional branch. This is primarily because GRAM and ORAM reveal random-looking access patterns to the parties. If an access comes from an inactive conditional SGC branch, then SGC's optimization will reveal information incompatible with the normal access pattern of the GRAM/ORAM. Thus, this use is insecure, as it allows the GC evaluator to identify the active branch in a conditional. See detailed discussion in Section 6. Other issues include the increased computational cost of processing GRAM's expensive access procedure in each branch. Similar concerns may apply to accessing other types of resources, such as stacks, queues, expensive procedure calls (e.g. non-black-box crypto primitives), or recent improved and unstackable GC techniques [HK21a].

In Section 6, we design a novel mechanism for efficiently and securely handling RAM accesses from within SGC branches. In short, our mechanism allows us to cheaply escape the conditional branch, access the resource, and then re-enter *that same branch*. Each branch can access a resource multiple times. Our mechanism allows fragments to access RAM without paying high cost for SGC gadgets.

We also note the following lower-level contributions:

Entire system and security proof. We package our approach as a garbling scheme and prove it secure.

Implementation. Our system is a non-trivial systems-technical undertaking.

3 RELATED WORK

In our review of related work, we focus on prior general purpose MPC tools.

Straight-line execution tools. The vast majority of MPC tools use straight-line execution, e.g. [RHH14, ZE15, DSZ15, WMK16, ACC+22, LHS+14]. These tools require that each program loop has a hard-coded upper bound. CBMC-GC goes one step further by trying to infer loop bounds automatically, but still ultimately models the program as a straight-line circuit [FHK+14].

Straight-line execution cannot suitably support arbitrary programs where the number of loop iterations depends on the data. We note that [HHNZ19] is an excellent systematization of knowledge that explores the pros and cons of such tools.

CPU emulation tools. We consider two works that operated in the CPU emulation paradigm [Kel17, WGMK16]. [Kel17] used SPDZ to implement a CPU-emulation-based protocol for malicious adversaries. While their online efficiency is competitive with the total cost of [WGMK16], their offline efficiency is $\approx 100\times$ slower. In our evaluation (Section 7), we accordingly focus our comparison on [WGMK16]. We described [WGMK16]'s approach in Section 1, and we compare to their performance in Section 7.

[WGMK16]'s uses Circuit ORAM [WCS15], which could be modularly swapped for a different ORAM, such as [Ds17], correspondingly affecting (improving) performance. We only compare to the existing system [WGMK16]. Constant-round complexity (and hence using EpiGRAM) is essential for CPU-emulation and VISA MPC due to the sequential nature of RAM accesses in these models. Interactive ORAMs incur latency cost proportional to the (large) number of steps of a typical program (cf. discussion in Section 7.4.3). Further, GRAM can be easily and cheaply upgraded to stronger security models, e.g. covert or malicious, using existing techniques. Such an upgrade for ORAM constructions, including [Ds17], is a challenge.

We note that TinyGarble implemented a MIPS ALU, but did not build on this to implement a working CPU emulation tool [SHS+15]. For example, they do not integrate RAM support to their prototype. Their main contributions are (1) better management of plaintext function by avoiding unrolling it into a plaintext circuit, and (2) applying hardware synthesis tools to reduce the size of the MIPS CPU, improving over naïve by up to 14.95%.

Loop coalescing. Loop coalescing is a compiler technique explored in the MPC context by [LWN+15] (and in the proof system context by [WSR+15]). The basic idea is to combine the bodies of loops into a single loop with an internal conditional. [LWN+15, WSR+15] show that this can improve MPC (resp. proof system) performance by reducing the number of hard-coded loop bounds in the program (cf. Section 1.1). The technique does not suggest (nor do [LWN+15, WSR+15] explore) further optimization, such as fragment design.

There are common characteristics of loop coalescing and VISA. Both techniques conditionally dispatch over program fragments.

Crucially, VISA approaches MPC optimization holistically, providing a clean abstraction and vocabulary for general optimization of oblivious programs (e.g. include stacking, GRAM, our new gadgets, etc.) and for expressing optimization constraints. Indeed, VISA emphasizes fragment design as a crucial optimization problem. VISA also provides a convenient vocabulary for discussing low level details, such as the size of a register file and managing the active set. See further discussion in Sections 5.3 and 5.4. In contrast, coalescing is a source code transformation, and is at the wrong level of abstraction for fragmentation and low-level optimization.

4 PRELIMINARIES

We implement our VISA machine using garbled circuits (GC). GC allows for powerful protocols that achieve secure computation in only a constant number of protocol rounds. We build on the half-gates GC technique [ZRE15], which requires that the parties communicate two ciphertexts per AND gate and zero ciphertexts per XOR gate [KS08].

We combine the basic [ZRE15] scheme with recent improvements in Garbled RAM [HKO21] and with Stacked Garbling [HK20, HK21b]. Garbled RAM is needed when accessing data from the VISA machine's main memory, and Stacked Garbling improves the communication consumption incurred when conditionally handling fragments.

We use these GC improvements heavily, and we overcome technical problems needed to compose them.

4.1 Garbled RAM

Compiling large arrays to Boolean circuits is infeasible. The problem is that on each array access, the circuit must touch each element of the array. Hence, on each access we pay cost proportional to the size of the array. Garbled RAM (GRAM) [LO13] equips GC with random-access arrays that incur only *sublinear cost*. GRAM preserves GC's important constant-round property.

A recent GRAM, called EpiGRAM, dramatically improved the concrete cost of the technique [HKO21]. We implemented EpiGRAM, and we use it to instantiate our VISA machine's main memory.

Our formalism manipulates GRAM directly by using two gates provided by EpiGRAM:

- An ARRAY gate takes as input public natural numbers n and w. It outputs a zero-initialized size-n array of width-w elements. We initialize all of our arrays width $w = 32$.
- An ACCESS gate performs an array access. The gate accepts as input (1) an array A, (2) $\log_2 n$ bits that together encode an array index α, (3) a bit rw that indicates if this is a read or a write, and (4) a w-bit value y that indicates what to store in the array if this is a write. As output, the gate yields (1) $A[\alpha]$ and (2) the updated array where the content of index α has been replaced by y iff $rw = 1$.

4.2 Stacked Garbling (SGC)

Until recent breakthrough work [HK20, HK21b], GC techniques required communication proportional to the computed program, including inactive branches. SGC [HK20, HK21b] achieves communication proportional to only the single longest *execution path* of the program.

This improvement is a boon to our approach, because we repeatedly conditionally evaluate the target program's fragments. SGC greatly improves the communication cost of fragments (see Section 7).

4.3 Cryptographic Assumptions

Our garbling scheme (Section 6.2) is secure under a typical GC assumption: We assume that the function H is a circular correlation robust hash function [CKKZ12, ZRE15].

As is standard in MPC (e.g., [GKK+12, WGMK16]), total runtime, i.e., the number of CPU emulation steps, is public. If desired, the steps can be padded.

We consider security in the presence of a semi-honest adversary. Since our construction is a garbling scheme, its security can be extended into covert, public verifiable covert (PVC), malicious models using standard techniques.

5 OUR VISA

The general idea of a VISA is agnostic of low-level details. Of course, it is interesting to instantiate and experiment with a specific architecture. We formalize our specific VISA here.

Our VISA is built on top of a *base ISA*. Our base ISA is indeed basic, providing primitive instructions that (1) perform algebraic operations, (2) achieve dynamic control flow, and (3) read/write main memory. We first formalize this base ISA. We choose a custom base ISA for simplicity of presentation and implementation; it may be desirable in future work to replace the base ISA with an off-the-shelf ISA, such as MIPS.

Once we establish the base ISA, we formalize our VISA, which essentially aggregates base instructions into fragments.

5.1 Base ISA

The base ISA specifies the instructions that can appear in our supported assembly programs. We emphasize that we do not execute these instructions one by one; rather, our VISA groups base instructions into fragments, and our VISA machine treats fragments as its atomic units of computation.

	Syntax	Semantics
Algebra	COPY *tar src*	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src]$
	CCOPY *tar* src_0 src_1	$\mathcal{R}[tar] \leftarrow \begin{cases} \mathcal{R}[src_1], & \text{if } \mathcal{R}[src_0] = 1 \\ \mathcal{R}[tar], & \text{otherwise} \end{cases}$
	ADD *tar* src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] + val(src_1)$
	SUB *tar* src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] - val(src_1)$
	MUL *tar* src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \cdot val(src_1)$
	XOR *tar* src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \oplus val(src_1)$
	AND *tar* src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \mathcal{R}[src_0] \wedge val(src_1)$
	EQ *tar* src_0 $\{src_1\}$	$\mathcal{R}[tar] \leftarrow \begin{cases} 1, & \text{if } \mathcal{R}[src_0] = val(src_1) \\ 0, & \text{otherwise} \end{cases}$
	CMP *tar* src_0 src_1	$\mathcal{R}[tar] \leftarrow 2 \cdot (\mathcal{R}[src_0] \overset{u}{<} \mathcal{R}[src_1]) + (\mathcal{R}[src_0] \overset{s}{<} \mathcal{R}[src_1])$
	SWAP src_0 src_1	$\mathcal{R}[src_0], \mathcal{R}[src_1] \leftarrow \mathcal{R}[src_1], \mathcal{R}[src_0]$
	RS1 *dst*	$\mathcal{R}[dst] \leftarrow \lfloor \mathcal{R}[dst]/2 \rfloor$
	IMM *dst imm*	$\mathcal{R}[dst] \leftarrow imm$
Control Flow	J *imm*	$\mathsf{pc} \leftarrow imm$
	JE *src imm*	$\mathsf{pc} \leftarrow \begin{cases} imm, & \text{if } \mathcal{R}[src] \neq 0 \\ \mathsf{pc}+1, & \text{otherwise} \end{cases}$
	JNE *src imm*	$\mathsf{pc} \leftarrow \begin{cases} imm, & \text{if } \mathcal{R}[src] = 0 \\ \mathsf{pc}+1, & \text{otherwise} \end{cases}$
	JL *src imm*	$\mathsf{pc} \leftarrow \begin{cases} imm, & \text{if } \mathcal{R}[src]\&1 \neq 0 \\ \mathsf{pc}+1, & \text{otherwise} \end{cases}$
	JB *src imm*	$\mathsf{pc} \leftarrow \begin{cases} imm, & \text{if } \mathcal{R}[src]\&2 \neq 0 \\ \mathsf{pc}+1, & \text{otherwise} \end{cases}$
	HALT	– no effect, pc unchanged –
Memory	LOAD *tar src*	$\mathcal{R}[tar] \leftarrow \mathcal{M}[\mathcal{R}[src]]$
	STORE *tar src*	$\mathcal{M}[\mathcal{R}[tar]] \leftarrow \mathcal{R}[src]$

$$val(x) \triangleq \begin{cases} x, & \text{if } x \text{ is an immediate} \\ \mathcal{R}[x], & \text{if } x \text{ is a register id} \end{cases}$$

Figure 2: Our base ISA. Each instruction type handles between zero and three arguments. In general, arguments refer to *registers*, but some arguments, denoted $\{\cdot\}$, can also optionally be *immediates* (i.e., compile-time constants). *val* is a helper function that resolves an argument that can be either a register or an immediate. Unless the semantics otherwise mention an effect on the pc, each instruction also increments the pc. The symbol $<$ with an overset u (resp. s) denotes a comparison where the arguments are treated as an unsigned (resp. signed) integers.

The base ISA formalizes both the syntax and the semantics of instructions. Our instructions each provide a simple mechanism for performing algebra, achieving control flow, or accessing memory. To define instruction semantics, we define an abstract machine that executes instructions. Our ISA simultaneously defines our instruction set and the abstract machine that runs them.

Definition 5.1 (Base ISA). Our instruction set is formally defined in Figure 2. The semantics of instructions are defined by reference to an abstract machine with a *program counter* pc, a *register file* $\mathcal{R}$, a main memory $\mathcal{M}$, and a *program* $\mathcal{P}$. pc is a 32-bit index that indicates which base instruction to execute next. $\mathcal{R}$ is a length-m array of 32-bit integers. $\mathcal{M}$ is a length-n array of 32-bit integers. $\mathcal{P}$ is an array of instructions. Both n and m are configurable parameters of the abstract machine. A machine is initialized with an arbitrary program. At initialization, pc, $\mathcal{R}$, and $\mathcal{M}$ are zero initialized. At each step, the machine updates itself based on the semantics of instruction $\mathcal{P}[\mathsf{pc}]$.

In our implementation, we instantiate a machine with a size-13 register file; we vary the size of RAM depending on the requirements of the executed program.

We emphasize that while both the register file and the memory are key-value data structures, our VISA machine handles them very differently. Our memory supports dynamic access and is implemented using Garbled RAM. On the other hand, our register file does not need to implement dynamic access: each usage of the register file is statically specified by an instruction, so each register is essentially just a named collection of 32 circuit wires. Inside a fragment, accessing the register file is free.

5.2 Fragments

As discussed and motivated in Section 2, batching multiple instructions by creating fragments resolves the bulk of the cost of the CPU emulation approach.

Definition 5.2 (Fragment). A *fragment* is a straight-line sequence of base ISA instructions where only the final instruction may be a Control Flow instruction (c.f. Figure 2).

Definition 5.2 coincides with the notion of a program basic block. We still elect to use new terminology because the notion of a fragment can be (and, we expect, will be) generalized, for example by allowing intra-fragment control flow. The only limitation in extending the above definition is that a fragment should never contain a data-dependent loop, since this would break the straight-line nature of the fragment. For simplicity, we do not explore this direction here, but we believe that this can be exploited heavily in future work.

We now define the syntax/semantics of our VISA.

Definition 5.3 (Our VISA). Like our base ISA, a VISA is a set of instructions together with the abstract machine that executes them. A VISA instruction is a *fragment* (Definition 5.2). The VISA abstract machine is identical to the base ISA machine, except that the program $\mathcal{P}$ consists of fragments, and at each step the machine executes the semantics of the current fragment $\mathcal{P}[\mathsf{pc}]$.

Remark 1. *Note, a VISA program is thus viewed as including the corresponding variable instruction set. A VISA then specifies the interpretation of the program. A VISA machine instantiates the (secure) execution of the program. While a full toolchain starts from programs written in a base ISA, the VISA definition is about programs that have been fragmented. In practice, the VISA machine toolchain will generate the fragmentation and hence the program's instruction set.*

While the above *specification* indicates an array lookup $\mathcal{P}[\mathsf{pc}]$, our instantiation dispatches fragments via conditional branching. Note that to achieve the prescribed semantics, we do not need to conditionally dispatch over *each* fragment at each step. In general, not all fragments will be possible at a given step. We reduce the number of conditionally dispatched fragments by considering a control flow graph (CFG) representation of the target program. We maintain a set of pointers into the CFG that indicates the set of possible pc values. At each step, our VISA machine only dispatches over those fragments that are currently pointed to.

5.3 Memory Hierarchy

A VISA introduces the opportunity to distinguish three types of memory:

- **Main memory.** Most program state is stored in a large main memory that is accessed dynamically at high cost.
- **Persistent registers.** The local state of a VISA machine is held in *persistent registers*. Inside the fragment, these registers are free. However, to conditionally dispatch over fragments, this local state must be passed to each branch. SGC imposes cost for each bit that crosses the interface to/from the conditional. It is sensible to store frequently used data in persistent registers, but the number of these registers should be kept in check.
- **Local registers.** Since register access is free inside a fragment, a VISA program can introduce arbitrary numbers of local registers, allowing the fragment to store a large state without paying for it. At the exit of the fragment, the content of local registers is lost.

Allocating data to these levels of memory is a large and interesting optimization space. We use 13 persistent registers and a RAM of size up to 2^{13} 32-bit words in our experiments.

5.4 Fragment Generation

As discussed in Section 2, the choice of strategy for breaking a program into fragments can dramatically affect performance. In this work, we align fragments with program basic blocks (i.e., each control flow instruction maps to a fragment), with one exception: we introduce extra fragments for RAM accesses such that each fragment has at most one RAM access. We found that this simple strategy reduces the overall number of RAM accesses[4], which we found is the performance bottleneck.

Note that for simplicity of presentation, Figure 1 does not show the extra fragments resulting from RAM accesses. Our actual fragmentation has 14 fragments.

While we leave further in-depth exploration of intelligently selecting fragments as significant future work, we outline several guidelines for such strategies. We note that these guidelines sometimes contradict one another, as fragment optimization is a challenging problem.

Generate fragments such that each conditional dispatch is over fragments of similar size and with a similar number of RAM accesses. SGC, and other approaches to MPC free branching [HKP20, HKP21], achieves communication cost proportional to the single most expensive branch. To best take advantage of free branching, ensure that branches have similar cost. This can be achieved, e.g., by splitting large program basic blocks into more than one fragment and/or by merging multiple basic blocks into a single fragment.

RAM access is an expensive resource; an unbalanced allocation across dispatched fragments misses an opportunity to amortize accesses.

Prefer larger fragments. This reduces the number of VISA machine steps. Hence, larger fragments further reduce the amount of CPU-emulation-style machinery.

Compress the interface to each fragment. As explained in Section 5.3, we pay to transport the content of persistent registers into and out of branches. Using compiler techniques to reduce the number of needed persistent registers will reduce cost.

Prefer fragmentation that leads to smaller active sets. SGC computational and interface costs scale with the number of branches, so we should seek to reduce the number of branches per step (i.e., to shrink each active set). One way this guideline might be achieved is by artificially introducing periodicity into a program's execution. For instance, we can split each loop into a number of fragments that is a power of two. Without periodicity in consecutive loops the active set will tend to grow with each step until it includes each program fragment. Artificially introducing periodicity groups

[4] I.e., all active set fragments will have a same number of accesses.

fragments into "congruence classes" and ensures that most fragments never coincide in the same active set. [WGMK16] considered a similar technique in their MIPS processor. Introducing periodicity for fragments introduces further opportunities to align code and amortize cost.

6 GAR: OUR VISA MACHINE

This section introduces GAR (Garbled Assembly with RAM) our implementation of the VISA machine. GAR is formalized as a garbling scheme [BHR12]. As already mentioned, GAR conditionally dispatches fragments using SGC and implements main memory via GRAM.

We first discuss technical issues and our solution in combining our two main building blocks, SGC and GRAM. Then, in Section 6.2 we present the GAR scheme and state the main security theorem (proofs are presented in the Appendix).

6.1 SGC with GRAM

The incompatibility of SGC and GRAM. SGC is compatible with many, but not all GC techniques. SGC requires that the string of material encoding each branch be indistinguishable from a uniform string. This restriction is needed to mask from the GC evaluator the identity of the conditional's active branch: if a branch is inactive, SGC arranges that the evaluator obtains uniform *garbage material*.

Unfortunately, GRAM's material *is distinguishable* from a uniform string. In short, GRAM will one-by-one reveal to the evaluator RAM indices that are randomly generated *without replacement* [HKO21]. These revealed indices are indistinguishable from a uniform permutation, but not from a uniform string. Thus it is not secure to use GRAM's ACCESS gate inside an SGC conditional.

SGC's uniform string requirement and GRAM's revealed uniform permutations seem somewhat inherent to the techniques, and it is not clear that we can revise these techniques to make them compatible with one another. Even if it were possible to make the two techniques compatible, it would not be desirable. SGC requires that each party garble each branch multiple times, introducing added computational cost. Since the GRAM access procedure is large, we would like to avoid repeatedly garbling it. It is more pragmatic to simply garble each access once, as we end up doing.

Our approach. One way we could handle RAM access in a VISA machine would be to place each RAM access instruction in its own single-instruction fragment. While correct and secure, the approach violates several of our guidelines for program fragmentation (Section 5.4), and is undesirable for a number of performance reasons. In particular, the resulting fragments are smaller, more numerous, and each RAM access will service a smaller fragment. Ultimately, this discards many of the VISA's benefits.

A much better way would be to temporarily escape a fragment just to perform the RAM access, then re-enter that fragment. This is the approach we take. We design a new scheme that allows us to temporarily escape an SGC conditional branch (i.e., a fragment), perform the access, then re-enter that same branch. Because we escape the SGC branch before accessing RAM, we avoid SGC's uniform string requirement. Thus, RAM access is simulatable. Crucially for performance, our gadgets escape, and not fully exit SGC, and transfer across the SGC interface only those specific bits that are directly related to the RAM access. Thus, we do not, for example, pay to transfer the full register file on each RAM access.

Instrumenting GRAM access in SGC. SGC uses two garbled gadgets, the *demux* and the *mux*, to enter and exit a conditional, respectively. Each of these gadgets handles branch input/output wire-by-wire, where each wire is (indirectly) connected from the outside of the conditional to the internal circuit of each branch. We refer to each of these wire connections as a *port* of the demux/mux. There is one port per external wire.

Our observation is that, in contrast with standard SGC, the demux/mux need not be evaluated in one shot at the very beginning/end of the conditional. Instead, the GC evaluator can process ports of the gadgets in an *arbitrary* order, so long as data dependencies in the circuit are satisfied.

This in particular means that the evaluator can (1) process input to a branch by handling only some ports of the demux, (2) evaluate some gates in that branch, generating input to a RAM query, (3) feed the RAM query through ports in the mux to temporarily escape the branch, (4) execute the RAM access outside of SGC, in plain GC, (5) feed the RAM result through ports of the demux back into the branch, and (6) continue evaluation of the branch.

Interestingly, the GC generator's order of building the corresponding GC material is different. Because each branch must be generated from a seed (this is a key trick behind SGC's improvement), the generator garbles each branch all at once, before any RAM accesses are handled. As part of doing so, he assigns uniformly random GC labels to the branch side of each demux port. Only once each branch is fully generated, does he generate GC for RAM access(es). Labels of these GCs match the labels of the ports of the SGC conditional. Finally, he generates the GC material for the demux and mux.

Our modification to SGC still uses the main ideas of Stacked Garbling [HK20]: our GC generator garbles each branch starting from a distinct PRG seed and then stacks the material together using XOR. Our GC evaluator can decrypt the seed for each inactive branch and hence can reconstruct their garblings, unstack the material for the active branch, and evaluate. I.e., our scheme retains the important communication advantage of SGC.

Next, we formalize our full garbling scheme GAR, which includes the above trick.

6.2 Our Scheme: Formalization and Theorems

We formalize our VISA machine as a *garbling scheme* [BHR12]. SGC [HK21b] and GRAM [HKO21] are also formalized as garbling schemes; our scheme reorganizes and adjusts their procedures, making them compatible with each other and with our VISA (Section 5).

At a high level, our scheme should be understood as a new SGC scheme equipped with black-box GRAM. As an aside, it is possible to replace black-box GRAM with other garbled resources, for example a stack or queue [ZE13].

Program description. A garbling scheme securely handles any program from some specified language. Our goal is to support programs expressed in our base ISA (Figure 2). At the lowest level, we have primitive support for AND gates [ZRE15], XOR

gates [KS08], SWITCH statements [HK21b], and ARRAY and ACCESS gates [HKO21]. The semantics of XOR and AND gates are natural; ARRAY and ACCESS gate semantics are specified in Section 4.1. A SWITCH executes only the indicated branch and outputs the result. We group instructions from our base ISA, then compile these to our low level primitives. Thus, our formal garbling scheme consists of three major steps:

- **Compile base ISA program to VISA program.** Our scheme first groups base ISA instructions into fragments using the strategy described in Section 5.4.
- **Compile VISA program to primitives.** We compile each fragment primitive operation using standard techniques. Each basic instruction has a corresponding straight-line circuit, and our scheme stitches together each of the circuits in the fragment. To conditionally dispatch, the scheme wraps the fragment circuits in a SWITCH.
- **Evaluate primitives via GC.** The most interesting step is the evaluation of primitives, which is explained below.

Note that the first two steps of our handling are quite modular. It is easy to replace the ISA to VISA compiler with one that, for example, more intelligently selects fragments. Similarly, we could replace the compiler from fragments to circuits with more sophisticated techniques. From here, our scheme focuses on the handling of primitives, which is its crypto-technical component.

Definition 6.1 (Primitive Circuit Program). A *primitive circuit program* is a circuit consisting of AND gates, XOR gates, ARRAY gates, ACCESS gates, and SWITCH statements. ARRAY and ACCESS gates are defined in Section 4.1. A SWITCH statement is recursively parameterized over b primitive circuit programs and $\lceil \log_2 b \rceil$ wires that indicate which branch to execute. Note that an ACCESS gate *is allowed* inside a SWITCH statement.

Our GAR scheme handles arbitrary assembly programs by appropriately implementing the above circuit primitives. We note that our assembly compiler generates restricted classes of circuit programs, and we need not handle them in full generality of Definition 6.1. For example, the resulting primitive program will not feature nested conditionals, and the ARRAY gate will be used exactly once to initialize main memory *MEM* at the start of the program. Furthermore, each ACCESS gate will be parameterized by the specific array *MEM*. Looking ahead, our formalism will handle only the relevant special forms of primitive circuit programs.

We are now ready to present our main construction, the GAR[5] (Garbled Assembly with RAM) garbling scheme [BHR12].

Construction 1 (GAR). *GAR consists of three components:*

- *A* fragmentation strategy *that specifies how to convert a base ISA program into a VISA program. GAR uses the strategy discussed in Section 5.4; we do not formally specify further.*
- *A* compiler *that transforms a VISA program (Definition 5.3) into a primitive circuit program (Definition 6.1); because each fragment has no data-dependent control flow, compiling each fragment to a primitive circuit program is straight-forward, and we do not specify further.*
- *A* garbling scheme *that securely executes primitive circuit programs (Definition 6.1).*

The GAR garbling scheme is the tuple of procedures:

$$(GAR.ev, GAR.Ev, GAR.Gb, GAR.En, GAR.De)$$

Note, GAR's functionality goes beyond simply instantiating a VISA machine; in particular GAR fragments programs written in base ISA and generates VISA programs. We could have treated this functionality separately as part of our toolchain.

We formally present procedures of the GAR garbling scheme in Appendix A, see Figures 10 to 13. Here, we review them at a high level.

GAR.ev. This procedure defines the semantics of primitives (Definition 6.1). The semantics of AND gates and XOR gates are natural. ARRAY and ACCESS gate semantics are specified in Section 4.1. A SWITCH executes only the indicated branch and outputs the result.

GAR.Ev. This procedure specifies the GC evaluator's handling. In short, the handling of primitives is inherited from prior work [ZRE15, HKO21]. The exception is our new SWITCH primitive, which supports RAM ACCESS inside its branches. We discussed our method for handling ACCESS gates from within a SWITCH in Section 6.1.

GAR.Gb. This procedure specifies the GC generator's handling. Again, the handling of primitives is inherited from prior work [ZRE15, HKO21]. See Section 6.1 for the handling of our SWITCH primitive.

GAR.En. This procedure specifies how cleartext GC inputs are mapped to GC labels. The procedure is standard: on each input wire, a zero maps to one label and a one maps to a different label.

GAR.De. This procedure specifies how output GC labels are mapped to cleartext outputs. The procedure is standard.

GAR meets the standard garbling scheme definitions of *correctness*, *authenticity*, *obliviousness* and *privacy* [BHR12]. Meeting these is sufficient to instantiate 2PC/MPC protocols. We state the definitions and prove that GAR meets them in the full version of this paper. Theorems in the full version imply the following:

Theorem 6.2 (Main). *Assuming a circular correlation robust hash function, GAR's garbling scheme is* correct, authentic, oblivious, *and* private.

7 EVALUATION

7.1 Implementation and Testing Environment

We implemented GAR and used it to instantiate a semi-honest 2PC protocol in ≈ 5200 lines of C/C++. We instantiated Oblivious Transfer and Network I/O using the EMP Toolkit [WMK16]. We ran our experiments on two `m6i.16xlarge`[6] machines in the same region of an Amazon EC2 cluster. One machine ran the GC generator and the other ran the GC evaluator. We also ran [WGMK16]'s implementation on the exact same setup to establish our baseline.

We configured both systems with the same inputs and with RAM of the same size. GAR handles the program written in our assembly language. [WGMK16] takes a MIPS binary compiled by an off-the shelf compiler, thus placing them somewhat at a disadvantage.

[5] A gar is a predatory and particularly menacingly looking fish.

[6] Intel(R) Xeon(R) Platinum 8375C CPU @ 2.90GHz, 256GiB Memory, 25Gbps Network

Program		# Mem Ent	Time (s)			Comm. (GB)			# RAM Accesses		
			[WGMK16]	Ours	Impr.	[WGMK16]	Ours	Impr.	[WGMK16]	Ours	Impr.
PSI	64	128	29.4	0.6	**49.0**×	0.7	0.1	**6.0**×	2455	256	**9.6**×
	256	512	311.7	7.1	**43.9**×	7.1	1.8	**3.9**×	9755	1024	**9.5**×
	1024	2048	4378.6	61.9	**70.7**×	102.0	17.1	**6.0**×	38855	4096	**9.5**×
Dijkstra's	40	1024	2601.3	62.6	**41.6**×	58.5	9.5	**6.2**×	21579	3902	**5.5**×
	60		5462.7	127.3	**42.9**×	127.0	18.5	**6.9**×	46679	8302	**5.6**×
	80		9723.0	216.5	**44.9**×	221.0	30.5	**7.2**×	81379	14202	**5.7**×
	100		14910.0	332.2	**44.9**×	341.0	47.0	**7.3**×	125779	21802	**5.8**×

Figure 3: Comparison of our GAR system with [WGMK16]. We run PSI and Dijkstra's for a range of input sizes. We ran both GAR and [WGMK16] on the same hardware setup. Our approach substantially improves wall-clock time, communication consumption, and RAM usage. Our count of [WGMK16]'s RAM accesses *does not* include instruction fetching; We list them separately in Figure 5.

G's Pattern Length	*E*'s String Length	GRAM Size (32-bits)	# Executed Fragments	Gb. Time(s)	Trans. Time(s)	Ev. Time(s)	Total Time(s)	#Inst.	Speed #Inst/s	#Mem Access	Comm. (GB)
50	400	512	1400	9.7	5.3	4.4	19.4	12535	646Hz	1401	2.53
150	700	1024	2700	21.2	15.4	9.8	46.4	23635	509Hz	2701	7.44
250	1500	2048	5500	49.6	47.1	23.1	119.8	48735	407Hz	5501	22.70
500	7000	8192	23000	275.3	358.7	131.3	765.3	209485	274Hz	23001	173.00

Figure 4: GAR's evaluation on KMP with different inputs and GRAM sizes.

7.2 Benchmarks and Metrics

As explained in Section 1 and further demonstrated in Section 7.4.5, straight-line execution is not feasible for programs with complex control flow. Accordingly, we focus comparison on [WGMK16]'s CPU emulation approach. We demonstrate significant improvement on three programs.

- **Private Set Intersection (PSI)**: Two parties each hold a sorted integer array and wish to compute the number of common elements. While fast tailored PSI protocols exist, we use this benchmark for direct comparison with prior work [WGMK16].
- **Dijkstra's shortest path (Dijkstra):** One party holds a directed graph while the other holds a pair of source and destination nodes (This is the setting of [WGMK16]; other input configurations, e.g., all inputs secret-shared, incur no extra cost). Parties wish to compute the shortest path between the two nodes. This benchmark was used by [WGMK16] and [LWN+15].
- **Knuth–Morris–Pratt string search (KMP):** One party inputs a pattern string and the other inputs a search string. They wish to compute the number of occurrences of the pattern in the search string. This benchmark was suggested by [LHS+14].

We note that our reported runtimes for [WGMK16] are in some cases slower than what was reported in [WGMK16] itself. We believe this is due to the fact that program runtime is variable and depends on the program input. Crucially, we ran our GAR system on the same input as [WGMK16], and thus our reported numbers are directly comparable.

Assembly code for each benchmark is included in the full version. We report the following metrics:

- **Wall-clock time:** Wall-clock time includes the time needed for the GC generator to garble, for network transmission, and for the GC evaluator to evaluate. Figure 6 provides a breakdown of these three metrics.
- **Communication:** Both [WGMK16] and GAR communicate through one TCP/IP connection. We directly measure communication from the TCP port and report the amount of transmitted data.
- **# RAM accesses:** RAM accesses are the most expensive operation in our approach. Recall that [WGMK16] uses ORAM while GAR uses GRAM. We report the number of times we and [WGMK16] access RAM. Recall that [WGMK16] uses RAM to fetch instructions; we do not. Figure 3 *does not* include [WGMK16]'s RAM accesses to fetch instructions. We list them separately in Figure 5.

7.3 Overall Improvement

Figure 3 tabulates GAR's improvement over [WGMK16] for PSI and Dijkstra's.

PSI. We ran the PSI benchmark on three different pairs of input arrays: two 64-element arrays, two 256-element arrays, and two 1024-element arrays. The PSI program primarily consists of a loop that compares a single element from each array. Our VISA approach captures PSI's loop in a *single fragment.* This results in simple control flow and high performance. In total, we use only three fragments: one that initializes state before the loop, one that implements the body of the loop, and one that handles the end of the program. Because the loop is captured by one fragment, our approach uses precisely the number of RAM accesses that are prescribed by the program's execution path.

GAR is 44–70× faster than [WGMK16] on each input, uses 4–6× less bandwidth, and uses ≈ 10× fewer RAM accesses.

Dijkstra's. We ran Dijkstra's (Figure 1) on a graph with 40, 60, 80, and 100 nodes. For a graph with n nodes, we set the number of

	PSI			Dijkstra			
Input Size	**64**	**256**	**1024**	**40**	**60**	**80**	**100**
Thousands of Fetches	2.5	9.8	38.9	21.6	46.7	81.4	125.8

Figure 5: [WGMK16]'s number of instruction fetch.

Bench.		**Gb. Time(s)**	**Trans. Time(s)**	**Ev. Time(s)**	**Total Time(s)**
PSI	64	0.2	0.3	0.1	0.6
	256	2.3	3.7	1.1	7.1
	1024	18.3	35.9	7.7	61.9
Dijkstra	40	29.4	19.5	13.7	62.6
	60	61.0	38.3	28.0	127.3
	80	103.6	62.9	50.0	216.5
	100	159.1	97.0	76.1	332.2

Figure 6: Breakdown of our total wall-clock time into the time needed to (1) garble the circuit, (2) transmit the GC across the network, and (3) evaluate the GC.

Program		**#Inst.**	**#Fragments**	**Impr.**	**Speed #Inst/s**
PSI	64	2419	129	**18.8**×	4032Hz
	256	9715	513	**18.9**×	1368Hz
	1024	38899	2049	**19.0**×	628Hz
Dijkstra's	40	33914	3900	**8.7**×	542Hz
	60	73714	8300	**8.9**×	579Hz
	80	128614	14200	**9.1**×	594Hz
	100	198814	21800	**9.1**×	598Hz

Figure 7: Comparison of the number of fragments with the number of base ISA instructions for PSI and Dijkstra's with different input sizes. The improvement ($\frac{\#\text{instructions}}{\#\text{fragments}}$) illustrates that we reduce the number of execution steps by an order of magnitude. Our Hz rate is base instructions per second.

edges $|E| = 3n$. The sparse graph is stored in the adjacency list. Our program is split into 14 fragments.

For each input, GAR is 42–45× faster than the baseline [WGMK16], uses 6–7× less bandwidth, and uses 5–6× fewer RAM accesses.

7.4 Performance Breakdown and Discussion

7.4.1 Breakdown of Wall-Clock Time. Figure 6 breaks down the wall-clock-time for each of our runs of PSI and Dijkstra. No one cost clearly stands out as the bottleneck. We note that we did not stream the GC from the generator to the evaluator; we expect that proper streaming would allow to overlap garbling and evaluation with transmission, essentially eliminating the separate cost of garbling and evaluation. We also include detailed GAR costs in Figure 4.

7.4.2 SGC Savings. Recall that we use SGC to stack GC material from fragments in the active set. In our Dijkstra experiments, we observed that SGC improved communication by roughly 3×, excluding the cost of GRAM access. We expect that this improvement will become more significant for larger and more complex programs

	Round-trip Delay		
	0ms	50ms	100ms
[WGMK16] (s)	314.5	11495.4 (+11180.9)	22984.8 (+22670.3)
GAR (s)	11.5	12.8 (+1.3)	14.5 (+3.0)
Improvement	27.3×	898.1×	1585.2×

Figure 8: Runtime comparison of GAR and [WGMK16] when solving PSI-256 for different latency settings. To tightly control latency, we ran these experiments on a single machine with a simulated (via the Linux `tc` command) 2Gbps network. For clarity, we note the added cost of latency in parentheses. Note, the sliding window in TCP implicitly forces latency-like delays on GAR.

G's **Pattern Length**	E's **String Length**	**EMP #AND Gates** ($\times 10^9$)	**EMP Total Time(s)**	**GAR Total Time(s)**	**Impr.**
50	400	0.142	**7.4**	19.4	0.4×
150	700	2.264	120.0	**46.4**	2.6×
250	1500	13.11	732.7	**119.8**	6.1×
500	7000	233.0	12843.3	**765.3**	16.8×

Figure 9: Comparison of GAR with EMP's straight-line execution on KMP. Our improvement over EMP increases with larger inputs. Note, EMP implements array lookup with linear scans, not GRAM. For very small arrays, linear scans outperform EpiGRAM, which explains EMP's performance in the smallest instance.

where the total number of fragments and likely the size of the active set will be larger.

7.4.3 Communication Rounds and Latency Impact. GAR is implemented via a garbling scheme, and our instantiation in the semi-honest model only requires performing (parallel) OTs and sending a single message from the generator to the evaluator. In contrast, [WGMK16]'s ORAM-based CPU uses multiple rounds of communication *per RAM access*.

This distinction is not significant in the ultra-low latency setup we have explored so far, but even modest latency harshly penalizes multi-round approaches. We evaluate the impact by executing one program (PSI-256) with various latencies. We used the Linux traffic control tool `tc` to configure a network with 2Gbps bandwidth and 0/50/100ms latencies. Figure 8 tabulates wall-clock time performance. (In this experiment we run both parties on a single machine, so measurements in Figure 8 are not identical to our other experiments.) With higher latency, GAR's execution speed is almost unchanged, but [WGMK16] becomes significantly slower; GAR's advantage grows from 27× on a 0ms latency network to 898× (resp. 1585×) with 50ms (resp. 100ms) latency.

7.4.4 Active Set Sizes. GAR (and VISA) performance declines with the increase of sizes of active sets (i.e., sets of fragments that can be possibly executed in the corresponding step). Let M be the total number of fragments. In our experiments we observe that the active set size starts with 1 and quickly grows to $M - 1$ as execution proceeds: $M = 4$ (resp. 15 and 11) for PSI (resp. Dijkstra and KMP). We view active set optimization as crucial future work.

7.4.5 Comparison with Straight-Line Circuit Evaluation. Finally, we illustrate the advantage of the VISA approach over straight-line circuit evaluation by comparing with the semi-honest 2PC of the widely adopted EMP Toolkit [WMK16]. We implemented the Knuth-Morris-Pratt (KMP) string-searching algorithm in both GAR and EMP. (We do not include [WGMK16]'s performance, because their repository did not include this benchmark.) Figure 9 tabulates the results, and Figure 4 presents a fine-grained analysis of GAR's performance results.

KMP searches for occurrences of a length-k pattern held by G in a length-m string held by E and outputs the number of occurrences. An important feature of KMP is its $O(m+k)$ time complexity, rather than the naive $O(m \cdot k)$. Circuits are not a suitable representation as KMP contains an inner loop that must be pessimistically unrolled a total of $O(m \cdot k)$ times when in fact only $O(m + k)$ total iterations are needed.

ACKNOWLEDGMENT

This work is supported in part by Cisco research award and NSF awards CNS-2246353, CNS-2246354, and CCF-2217070. This material is also based upon work supported in part by DARPA under Contract No. HR001120C0087. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

REFERENCES

[ACC+22] Abdelrahaman Aly, Benjamin Coenen, Kelong Cong, Karl Koch, Marcel Keller, Dragos Rotaru, Oliver Scherer, Peter Scholl, Nigel P. Smart, Titouan Tanguy, and Tim Wood. SCALE-MAMBA software. https://homes.esat.kuleuven.be/~nsmart/SCALE/, 2022.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

[CKKZ12] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. On the security of the "free-XOR" technique. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 39–53. Springer, Heidelberg, March 2012.

[Ds17] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 523–535. ACM Press, October / November 2017.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015.

[FHK+14] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Cbmc-gc: An ansi c compiler for secure two-party computations. In Albert Cohen, editor, *Compiler Construction*, pages 244–249, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 513–524. ACM Press, October 2012.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[HHNZ19] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy*, pages 1220–1237. IEEE Computer Society Press, May 2019.

[HK20] David Heath and Vladimir Kolesnikov. Stacked garbling - garbled circuit proportional to longest execution path. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 763–792. Springer, Heidelberg, August 2020.

[HK21a] David Heath and Vladimir Kolesnikov. One hot garbling. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 574–593. ACM Press, November 2021.

[HK21b] David Heath and Vladimir Kolesnikov. LogStack: Stacked garbling with $O(b \log b)$ computation. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part III*, volume 12698 of *LNCS*, pages 3–32. Springer, Heidelberg, October 2021.

[HKO21] David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. Practical garbled RAM: GRAM with $O(\log^2 n)$ overhead. Cryptology ePrint Archive, Report 2021/1519, 2021. https://eprint.iacr.org/2021/1519.

[HKP20] David Heath, Vladimir Kolesnikov, and Stanislav Peceny. MOTIF: (almost) free branching in GMW - via vector-scalar multiplication. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 3–30. Springer, Heidelberg, December 2020.

[HKP21] David Heath, Vladimir Kolesnikov, and Stanislav Peceny. Masked triples - amortizing multiplication triples across conditionals. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 319–348. Springer, Heidelberg, May 2021.

[HYDK21] David Heath, Yibin Yang, David Devecsery, and Vladimir Kolesnikov. Zero knowledge for everything and everyone: Fast ZK processor with cached ORAM for ANSI C programs. In *2021 IEEE Symposium on Security and Privacy*, pages 1538–1556. IEEE Computer Society Press, May 2021.

[Kel17] Marcel Keller. The oblivious machine - or: How to put the C into MPC. In Tanja Lange and Orr Dunkelman, editors, *LATINCRYPT 2017*, volume 11368 of *LNCS*, pages 271–288. Springer, Heidelberg, September 2017.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Heidelberg, July 2008.

[LHS+14] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating efficient RAM-model secure computation. In *2014 IEEE Symposium on Security and Privacy*, pages 623–638. IEEE Computer Society Press, May 2014.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 523–542. Springer, Heidelberg, August 2018.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 719–734. Springer, Heidelberg, May 2013.

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. IEEE Computer Society Press, May 2015.

[RHH14] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy*, pages 655–670. IEEE Computer Society Press, May 2014.

[SHS+15] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE Computer Society Press, May 2015.

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 850–861. ACM Press, October 2015.

[WGMK16] Xiao Shaun Wang, S. Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure computation of MIPS machine code. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016, Part II*, volume 9879 of *LNCS*, pages 99–117. Springer, Heidelberg, September 2016.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[WSR+15] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS 2015*. The Internet Society, February 2015.

[YHKD22] Yibin Yang, David Heath, Vladimir Kolesnikov, and David Devecsery. Ezee: Epoch parallel zero knowledge for ansi c. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 109–123. IEEE, 2022.

[YPHK23] Yibin Yang, Stanislav Peceny, David Heath, and Vladimir Kolesnikov. Towards generic mpc compilers via variable instruction set architectures (visas). Cryptology ePrint Archive, Paper 2023/953, 2023. https://eprint.iacr.org/2023/953.

[ZE13] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In *2013 IEEE Symposium on Security and Privacy*, pages 493–507. IEEE Computer Society Press, May 2013.

[ZE15] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. https://eprint.iacr.org/2015/1153.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.

A FULL GARBLING SCHEME

We provide the formal procedures for GAR. Figure 12 lists the scheme procedures (i.e., Construction 1) of GAR. Figure 13 explains how we handle ACCESS gates internal to SGC branches. Figure 10 and Figure 11 are unrolled modifications of the COND gate procedures from LogStack.

In Definition 6.1, we define primitive programs as having explicit AND gates, XOR gates, etc. For brevity and to closely match the procedures of [HK21b], which are conceptually quite similar, we use slightly different syntax our figures. I.e., a *netlist* is a sequence of AND and XOR gates. Netlists are handled via the [ZRE15] garbling scheme. 'Cond' statements correspond to the SWITCH keyword. 'Seq' statements denote two circuit components that are run in sequence. We emphasize that this language-level difference does not change the meaning of primitive circuit programs.

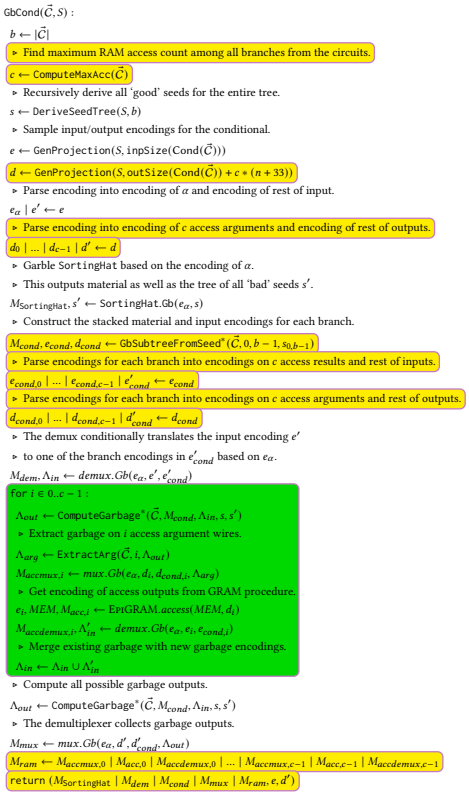

Figure 10: The algorithm for garbling a conditional with b branches where each branch has at most c RAM accesses. Main memory is a length-2^n GRAM with 32-bit entries. GbCond follows the structure of LogStack's procedure of the same name. Our colored boxes highlight diffences as compared to LogStack, and the green box highlights the most important modification.

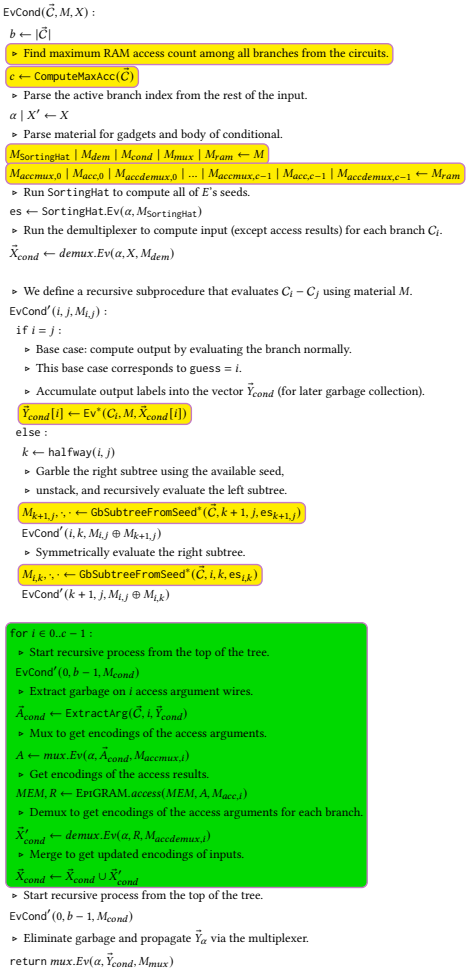

Figure 11: E's procedure, EvCond**, evaluates a conditional with b branches and at most c RAM accesses.** Our procedure follows the structure of LogStack's procedure of the same name. Colored boxes highlight the major differences as compared to LogStack, and the green box highlights the most important modification.

```
GAR.ev(C, x⃗) :
  ▷ m⃗ is a global array initialized to all 0.
  ▷ What are the circuit semantics?
 Switch C :
  case Netlist(·) : return HalfGates.ev(C, x⃗)
  case Seq(C_0, C_1) : return GAR.ev(C_1, GAR.ev(C_0, x⃗))
  case Cond(C⃗) :
    ▷ split branch index from input
    α | x⃗' ← x⃗
    ▷ Run the active branch.
    return GAR.ev(C⃗[α], x⃗')
  case Access(idx⃗, val⃗, rw, dst⃗) :
    if x⃗[rw] == 0 :
      ▷ Load
      x⃗[dst⃗] ← m⃗[x⃗[idx⃗]]
    else :
      ▷ Store
      m⃗[x⃗[idx⃗]] ← x⃗[val⃗]
    return x⃗

GAR.Ev(C, M, X⃗) :
  ▷ MEM is a unique global Array object by EpiGRAM.init(·)
  ▷ How does E evaluate the GC?
 Switch(C) :
  case Netlist(·) : return HalfGates.Ev(C, M, X⃗)
  case Seq(C_0, C_1) :
    M_0 | M_tr | M_1 ← M
    return GAR.Ev(C_1, M_1, trans.Ev(GAR.Ev(C_0, M_0, X⃗), M_tr))
  case Cond(C⃗) : return EvCond(C⃗, M, X⃗)
  case Access(idx⃗, val⃗, rw, dst⃗) :
    ▷ Call GRAM access procedure
    MEM, Y⃗ ← EpiGRAM.access(MEM, X⃗[idx⃗], X⃗[val⃗], X⃗[rw], M)
    X⃗[dst⃗] ← Y⃗
    return X⃗

GAR.Gb(1^κ, C, S)
  ▷ MEM is a unique global Array object by EpiGRAM.init(·)
  ▷ How does G garble the GC?
  ▷ S is an explicit seed.
 Switch C :
  case Netlist(·) :
    return HalfGates.Gb(1^κ, C, S)
  case Seq(C_0, C_1) :
    ▷ Derive seeds for two circuits.
    S_0 ← F_S(0)
    S_1 ← F_S(1)
    (M_0, e_0, d_0) ← GAR.Gb(1^κ, C_0, S_0)
    (M_1, e_1, d_1) ← GAR.Gb(1^κ, C_1, S_1)
    ▷ Labels out of C_0 must be translated
    ▷ to labels into C_1.
    M_tr ← trans.Gb(d_0, e_1)
    M ← M_0 | M_tr | M_1
    return (M, e_0, d_1)
  case Cond(C⃗) : return GbCond(C⃗, S)
  case Access(idx⃗, val⃗, rw, dst⃗) :
    ▷ Generate encodings for access arguments
    e_idx⃗ ← repeatedly-sample F_S(·)
    e_val⃗ ← repeatedly-sample F_S(·)
    e_rw ← repeatedly-sample F_S(·)
    ▷ Call GRAM access procedure
    d, MEM, M_acc ← EpiGRAM.access(MEM, e_idx⃗, e_val⃗, e_rw)
    return (M_acc, e_idx⃗ | e_val⃗ | e_rw, d)
```

Figure 12: GAR's garbling scheme. The included algorithms are typical except for the handling of conditionals. *Ev* and *Gb* delegate the core of our approach: EvCond (Figure 11) and GbCond (Figure 10). *En* and *De* are not listed as they are standard.

```
GAR.Ev*(C, M, X⃗) :
  ▷ Similar as Ev until access case
  case Access(idx⃗, val⃗, rw, dst⃗) :
    M_acc | M_tr ← M
    ▷ The access happened outside conditional
    ▷ M should be ⊥
    if X⃗ has dst⃗ ready : return X⃗
    else : ▷ HALT and return X⃗ to top layer

GAR.Gb*(1^κ, C, S)
  ▷ Similar as Gb until access case
  case Access(idx⃗, val⃗, rw, dst⃗) :
    ▷ Generate encodings
    e, d ← repeatedly-sample F_S(·)
    ▷ Treat it as inputs and outputs
    return (⊥, e, d)
```

Figure 13: Variants for *Gb* and *Ev*. These variants are called inside GbCond and EvCond. We implicitly use *function** to denote *function* from where the underlying calling to *Gb* (resp. *Ev*) is replace by *Gb** (resp. *Ev**).