# Efficient Maliciously Secure Multiparty Computation for RAM

Marcel Keller[1](✉) and Avishay Yanai[2]

[1] University of Bristol, Bristol, UK
M.Keller@bristol.ac.uk
[2] Bar-Ilan University, Ramat Gan, Israel
Ay.Yanay@gmail.com

**Abstract.** A crucial issue, that mostly affects the performance of actively secure computation of RAM programs, is the task of reading/writing from/to memory in a private and authenticated manner. Previous works in the active security and multiparty settings are based purely on the SPDZ (reactive) protocol, hence, memory accesses are treated just like any input to the computation. However, a garbled-circuit-based construction (such as BMR), which benefits from a lower round complexity, must resolve the issue of converting memory data bits to their corresponding wire keys and vice versa.

In this work we propose three techniques to construct a secure memory access, each appropriates to a different level of abstraction of the underlying garbling functionality. We provide a comparison between the techniques by several metrics. To the best of our knowledge, we are the *first* to construct, prove and implement a concretely efficient garbled-circuit-based actively secure RAM computation with dishonest majority.

Our construction is based on our third (most efficient) technique, cleverly utilizing the underlying SPDZ authenticated shares (Damgård et al., Crypto 2012), yields lean circuits and a constant number of communication rounds per physical memory access. Specifically, it requires no additional circuitry on top of the ORAM's, incurs only two rounds of broadcasts between every two memory accesses and has a multiplicative overhead of 2 on top of the ORAM's storage size.

Our protocol outperforms the state of the art in this settings when deployed over WAN. Even when simulating a very conservative RTT of 100 ms our protocol is at least one order of magnitude faster than the current state of the art protocol of Keller and Scholl (Asiacrypt 2015).

This research was supported by a grant from the Ministry of Science, Technology and Space, Israel, and the UK Research Initiative in Cyber Security. This work has been supported in part by EPSRC via grants EP/M012824 and EP/N021940/1, by the European Research Council under the ERC consolidators grant agreement n. 615172 (HIPS) and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

J. B. Nielsen and V. Rijmen (Eds.): EUROCRYPT 2018, LNCS 10822, pp. 91–124, 2018.
https://doi.org/10.1007/978-3-319-78372-7_4

# 1 Introduction

## 1.1 Background

Actively secure multiparty computation (in the dishonest majority setting) allows $n$ parties to compute an arbitrary function over their private inputs while preserving the privacy of the parties and the correctness of the computation even in the presence of a malicious adversary, who might corrupt an arbitrary strict subset of the parties.

The field of secure two-party (2PC) and multiparty (MPC) computation has a rich literature, starting with Yao [40] and Goldreich-Micali-Wigderson [16] and attracted much interest during the past decade due to advances in efficiency and fast implementations [8,23,26,37,38]. Nevertheless, almost all previous works require the parties to first "unroll" the function into an arithmetic or Boolean circuit representation and then securely evaluate the circuit gate by gate. This is in contrast to modern design of algorithms of practical interest (e.g., binary search, Dijkstra's shortest-path algorithm, Gale-Shapley stable matching, etc.) that are typically represented as Random Access Machine (RAM) programs that contain branches, recursions, loops etc., which utilize the $O(1)$ access to memory, rather than circuits. In the following we provide the necessary overview on the RAM model of computation and how it is securely realized.

**RAM Model of Computation.** RAM is classically modeled as a protocol that is carried out between two entities: $\mathsf{CPU}$ and $\mathsf{MEMORY}$, which are essentially a couple of polynomial time Turing machines, such that their storage capacity is unbalanced, specifically, the $\mathsf{CPU}$ usually stores a small amount of data, corresponding to the state of the program, which is logarithmic in the amount of storage in $\mathsf{MEMORY}$ required by the program. We denote the $\mathsf{CPU}$'s storage by $d$ and the $\mathsf{MEMORY}$'s storage by $D$ such that $|D| = N$ and $|d| = O(\log N)$. We denote a memory block at address $i$ by $D[i]$. During the program execution $\mathsf{CPU}$ typically chooses to perform one instruction $I$ out of a final instructions set $IS$. A program $\Pi$ and an input $\boldsymbol{x}$ are first loaded into the storage of $\mathsf{MEMORY}$ and then the $\mathsf{CPU}$ is being triggered to start working. From that point, $\mathsf{CPU}$ and $\mathsf{MEMORY}$ are engaged in a protocol with $T$ rounds where $T$ is the running time of $\Pi$. In the $t$-th round:

1. $\mathsf{CPU}$ computes the *CPU-step function*:
$$\mathrm{C}_{\mathrm{CPU}}(\mathsf{state}_t, b_t^{\mathsf{read}}) = (\mathsf{state}_{t+1}, i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, b_t^{\mathsf{write}}) \tag{1}$$
by executing instruction $I_t \in IS$. The input $\mathsf{state}_t$ is the current state of the program (registers etc.), $b_t^{\mathsf{read}}$ is the block that was most recently loaded from $\mathsf{MEMORY}$. The outputs of the CPU-step are: The new program's state $\mathsf{state}_{t+1}$, the address $i_t^{\mathsf{read}}$ in $D$ to read from and the address $i_t^{\mathsf{write}}$ in $D$ to write the block $b_t^{\mathsf{write}}$ to.
2. $\mathsf{CPU}$ sends $(i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, b_t^{\mathsf{write}})$ to $\mathsf{MEMORY}$. We define $\mathsf{access}_t \triangleq (i_t^{\mathsf{read}}, i_t^{\mathsf{write}})$.
3. $\mathsf{MEMORY}$ sends data block $D[i_t^{\mathsf{read}}]$ to $\mathsf{CPU}$ and assigns $D[i_t^{\mathsf{write}}] = b_t^{\mathsf{write}}$.

In every such a round, $\mathsf{CPU}$ is said to make a single request, or *logical access*, to $\mathsf{MEMORY}$. The output of the protocol, denoted $\boldsymbol{y} = \Pi(D, \boldsymbol{x})$, is the result of the computation of the program $\Pi$ on input $\boldsymbol{x}$ and memory $D$, such that $\mathsf{CPU}$ sets $\boldsymbol{y}$ as the last state of the program, $\mathsf{state}_{T+1}$. The sequence of accesses $\{\mathsf{access}_1, \ldots, \mathsf{access}_T\}$ is called the *access pattern of $\Pi$ on input $\boldsymbol{x}$ and memory $D$ (of size $N$)* and denoted $\mathsf{AP}(\Pi, D, \boldsymbol{x})$. Similarly, the sequence $\{I_1, \ldots, I_T\}$ is called the *instruction pattern* and denoted $\mathsf{IP}(\Pi, D, \boldsymbol{x})$.

The general methodology of designing secure multiparty computation directly to RAM programs is by having the parties take both the role of $\mathsf{CPU}$ and $\mathsf{MEMORY}$ and sequentially evaluate sufficiently many copies of the $\mathrm{C}_{\mathrm{CPU}}$ function. Upon completing the evaluation of one function, the parties access $D$ according to $\mathrm{C}_{\mathrm{CPU}}$'s output $(i_t^{\mathsf{read}}, i_t^{\mathsf{write}}, b_t^{\mathsf{write}})$ and obtain the input $b_t^{\mathsf{read}}$ to the next function.

Obviously, a secure protocol must not reveal $D$ to the parties, otherwise it would be possible to learn information about the parties' inputs. Trivially avoiding this is by embedding two sub-procedures inside $\mathrm{C}_{\mathrm{CPU}}$, one to encrypt (and authenticate) $b^{\mathsf{write}}$ before it is output and one to decrypt (and verify authentication of) $b^{\mathsf{read}}$ before it is used by $\mathrm{C}_{\mathrm{CPU}}$. This enhanced function is denoted $\mathrm{C}_{\mathrm{CPU}^+}$. Let $\mathrm{C}^1_{\mathrm{CPU}^+}, \ldots, \mathrm{C}^T_{\mathrm{CPU}^+}$ be garbled versions of $\mathrm{C}_{\mathrm{CPU}^+}$. The parties feed their inputs $\boldsymbol{x} = x^1, \ldots, x^n$ into $\mathrm{C}^1_{\mathrm{CPU}}$, taking the place of the wires associated with $\mathsf{state}_1$ and sequentially evaluate the garbled circuits to obtain $\boldsymbol{y} = \mathsf{state}_{T+1}$. This way, even an adversary who can tap (or even tamper) the memory accesses is unable to manipulate the program so it operates over forged data (since the data blocks are authenticated), yet, it might reveal information about the parties' inputs or program's state from the access pattern.

**ORAM in Secure RAM Computation.** Previous works on 2PC and MPC for RAM programs [1,11,14,18–20,22,24,28–31,39] use Oblivious RAM (ORAM) as an important building block. Informally speaking, an ORAM scheme is a technique to transform a program $\Pi$ with runtime $T$ and initial storage $D$ to a new program $\Pi'$ with runtime $T'$ and initial storage $D'$ such that the access pattern $\mathsf{AP}(\Pi', D', \boldsymbol{x})$ appears independent of both $\Pi$ and $\boldsymbol{x}$, yet, both programs compute the same function, i.e. $\Pi(D, \boldsymbol{x}) = \Pi'(D', \boldsymbol{x})$ for all $\boldsymbol{x}$. All ORAM schemes that we know of work by first initializing the storage $D$ and then *online simulate* each memory access individually (i.e. we don't know of a scheme that simulates a bunch of accesses altogether). It was shown feasible, since the work of Goldreich and Ostrovsky [17], that the simulation of a single memory access of $\Pi$ (which denoted by *logical access* above) incurs $\mathsf{poly}(\log N)$ memory accesses in $\Pi'$, denoted *physical accesses*, which leads to the same run time overhead, that is $T' = T \cdot \mathsf{poly}(\log N)$. In addition, we can obtain the same overhead for memory consumption of $\Pi'$, that is $N' = N \cdot \mathsf{poly}(\log N)$.

The general methodology for secure computation of RAM programs using an ORAM scheme is by having the parties collaboratively compute an ORAM transformation of $\Pi$ and $D$ (via any MPC protocol) to obtain $\Pi'$ and $D'$. This is a one-time step that incurs a computational and communication complexity that is proportional to $N'$. Then, they engage in a protocol of $T'$ steps to compute $\Pi'(D', x)$ and, as before, obtain the output as $\mathsf{state}_{T'+1}$.

This way, to securely compute a program, it is no longer required to unroll it to a circuit, rather, it is enough to unroll only the ORAM scheme algorithms and the CPU-step function. Consequently, this approach may lead to concentrated research efforts to optimize a specific set of ORAM scheme algorithms instead of looking for optimizations to the circuit version of each individual program.

**Oblivious vs. Non-oblivious Computation.** We distinguish between oblivious and non-oblivious computation in the following sense: In *oblivious computation* the parties learn nothing about the computation (except its output and runtime). Specifically, the parties learn nothing about either the program $\Pi$, $\mathsf{CPU}$'s state or the input $\boldsymbol{x}$. This means that an oblivious computation is applicable for private function evaluation (PFE) in which the function itself is kept secret. On the other hand, *non-oblivious computation* allows the parties to learn which instruction is being computed in which time step, in particular, it rules out algorithms that branch on secret values (otherwise, information about the secret values might be leaked). As noted in [28], in order to hide the instruction being computed - in every time step every possible instruction must be executed. The only implementation of an oblivious computation with active security that we know of is by Keller [22]. It has a performance of 41 Hz (physical memory accesses per second) in the online phase with $1024 \times 64$ bit memory and 2 Hz for $2^{20} \times 64$ bit memory[1] for 2 parties running over a local network. On the other hand, secure non-oblivious computation (denoted "instruction-trace oblivious" in [28]) is expected to yield a much better throughput, since the parties can avoid securely evaluating the universal CPU-step circuit, but can instead simply evaluate a much smaller circuit corresponding to the current instruction.

Notwithstanding the theoretical results in this paper hold for oblivious computation, the implementation results we report hold only for the non-oblivious settings. This relaxation is justified by the fact that non-oblivious computation is applicable for plenty of useful algorithms such as graph and search algorithms.

**Achieving Efficient Protocols.** To achieve an efficient, actively secure RAM computation the following crucial issues are to be addressed:

1. *Round complexity.* As explained above, securely evaluating a program requires $T' = T \cdot \mathsf{poly}(\log N)$ rounds of interaction between $\mathsf{CPU}$ and $\mathsf{MEMORY}$, corresponding to the $T'$ physical memory accesses. Also note that the access pattern of a program is determined by the input that it is given. Now, consider the CPU-step at time $t$ from Eq. (1), the parties need to read $D[i_t^{\mathsf{read}}]$ and map it to the input wires associated with $b_t^{\mathsf{read}}$. However, they do not know from ahead (i.e. when garbling) which address $i_t^{\mathsf{read}}$ would be accessed in which timestep and thus cannot map the input wire labels of $\mathrm{C}_{\mathrm{CPU}}^t$ to the

[1] The decrease in throughput reflects the runtime overhead implied by the ORAM, as mentioned above, this overhead depends on the memory size $N$.

right memory location. Therefore, achieving a protocol with round complexity independent in $T$ is much more challenging, in fact, there is a line of works that proposes constant-round secure RAM computation [7,11–14,19,30], however, it is highly impractical. A more reasonable path, which we follow in this paper, is to construct a scheme with a constant number of rounds *per any number of parallel physical memory accesses*. Although there exist *passively* secure implementations [28,29] that are constant-round per physical memory access, the *actively* secure implementations that we know of [22,24] have a round complexity linear in the depth of the CPU-step circuit (which depends on the ORAM implementation).

2. *Private and authenticated memory.* A natural approach suitable for securely handling memory is to choose an ORAM that encrypts its memory contents. In this approach, the parties must evaluate CPU-step circuits that include encryption/decryption and authentication/verification sub-circuits. This is undesirable since the resulting construction is non black-box in its underlying encryption/authentication primitives and, more practically, embedding encryption and authentication sub-circuits in every CPU-step circuit adds a large overhead in terms of computation, communication and space complexities for garbling and evaluating. This would be especially objectionable when the original RAM's behavior is non-cryptographic. The circuitry overhead when using the sub-circuit approach is demonstrated for several memory sizes in Table 1 where the circuit size is for a typical instruction that requires memory access[2]. *Circuit size* refers to the number of AND gates in the circuit performing a logical access, *read* and *write* are the number of bits being accessed and *encryption/authentication size* is the number of AND gates that would be necessary when incorporating the encryption and authentication procedures inside the CPU-step circuit. We measure the overhead using both our technique (described in Sect. 4.1) and a trivial solution using the AES block cipher (with circuit size of 6000 AND gates[3]), assuming blocks of $s = 40$ bits. We can see that even with our improvement (due to SPDZ representation of memory), securing memory accesses incurs an additional circuitry that is about 45 times larger than the ORAM circuit itself, therefore, we are highly motivated to find other techniques for transferring memory from storage to circuits.
3. *Memory consumption.* In actively secure BMR-based protocols memory used for storing the garbled circuit grows linearly with the number of gates and the number of parties. Let $G$ be the number of AND gates, $n$ the number of participants and $\kappa$ the security parameter. In the online phase each party stores $4 \cdot G \cdot n \cdot \kappa$ bits that represent the garbled circuit and additional $2 \cdot G \cdot \kappa$ bits that represent its own keys (the latter are needed to verify authenticity of the keys revealed during the evaluation and deciding which garbled entry to use next). For example, the SHA1 circuit is composed of $\sim$236K gates,

[2] The amount of memory being accessed to satisfy a CPU instruction depends on the instruction itself, for instance a SIMD instruction access more data than a SISD instruction.

[3] The state-of-the-art construction of an AES circuits incurs only 5200 AND gates.

among them ∼90K are AND gates. The evaluation of SHA1 with $\kappa = 128$ by 3 parties incurs memory of size ∼160 Mb and 0.5 Gb when evaluated by 10 parties. While this amount is manageable for a single execution of a circuit, it is much harder to be maintained when $T$ garbled circuits are evaluated sequentially in an online phase, as needed in RAM computation. Thus, new techniques must be developed to address that issue.

**Table 1.** Proportion of additional circuitry, counted as the number of additional AND gates, for the purpose of memory encryption and authentication in a logical memory access.

| Mem size | Circuit size | Read | Write | Enc/Auth. via technique, Sect. 4.1 | Enc/Auth. via block cipher (AES) |
|---|---|---|---|---|---|
| $2^{13}$ | 94844 | 31058 | 29596 | 2426160 (2600%) | 18192000 (19000%) |
| $2^{17}$ | 156990 | 92568 | 87634 | 7208080 (4500%) | 54060000 (34400%) |
| $2^{21}$ | 269300 | 158508 | 147982 | 12259600 (4500%) | 91944000 (34100%) |
| $2^{25}$ | 423014 | 249104 | 231098 | 19208080 (4500%) | 144060000 (34000%) |

## 1.2 Our Contribution

We construct and implement the first actively secure, garbled-circuit-based ORAM multiparty protocol. Specifically, we present the following contributions:

1. **Efficient Secure Memory Access.** We propose and compare three techniques to implement memory access in a secure computation for RAM programs. We briefly describe them in an increasing order of efficiency:
   (a) In the first technique, for each memory data item each party stores a SPDZ share of that data item. We stress that this technique has nothing to do with the SPDZ protocol, it only uses SPDZ shares representation to represent the memory content. In each access the data item is being re-shared using fresh randomness from the parties. Since SPDZ shares are also authenticated we achieve an authenticated memory as well. The re-sharing procedure is implemented as a sub-circuit, using only 2 field multiplications, which are embedded in every CPU-step circuit. For each $s$-bit block being accessed, the parties need to communicate $O(sn^2\kappa)$ bits (by all parties together) to reveal the appropriate keys for the input wires in the next CPU-step circuit, where $\kappa$ and $s$ are the computational and statistical security parameters respectively, $s$ is also the size of a SPDZ share. This is because every CPU-step receives $n$ shares, each of size $s$ bits, and for every bit all parties need to broadcast their keys of size $\kappa$ bits. The technique requires two rounds of broadcast per physical memory access, however, as explained above, embedding encryption and authentication sub-circuits has theoretical and practical disadvantages.

(b) The second technique is inspired by [1,32], in which the memory is implemented via wire soldering. That is, since every wire already carries a hidden and authentic value through the key that is revealed to the parties, the key itself could be stored in memory. This way, the parties do not need to transform wire keys to data items back and forth for every access, instead, they use wire soldering directly from the "writing circuit" to the "reading circuit". This "prunes away" the additional circuitry of the first technique, with the drawback of having each bit in the ORAM memory represented using a BMR key, i.e. $n\kappa$ bits (with $n$ the number of parties and $\kappa$ the security parameter). This technique, however, is superior in the other metrics as well, that is, it requires much less triples to be generated in the offline phase since it does not need the additional circuitry (which includes many AND gates) and has 2 communication rounds for each physical memory access, just like the first technique.

Naively generalizing the soldering of [1] to the *multiparty* settings requires each party to commit to its keys to all other parties using a xor-homomorphic commitment scheme. Instead, in this work we obviate the need of a commitment scheme and show how to use the readily available keys' shares. Moreover, we show how to do that black-box in the BMR garbling functionality, even when using the Free-XOR optimization [2], by which different garbled circuits are assigned with *different global differences.*

(c) The third technique offers a clever improvement to the soldering in that we solder only *one bit*, namely, the real value that passes through a wire, instead of the whole key that represents that value. As such, the soldering requires *no offline overhead at all*, that is, in contrast to the second technique, this technique does not invoke the multiplication command of the underlying MPC. We utilize the fact that the BMR-evaluation procedure reveals to the parties the external bit of each output wire (that is associated with a bit to be written to memory) and the fact that the permutation bits are already shared. This way the parties could obtain a share to a single bit which is the XOR (addition in the binary field) of the external and permutation bits.

Nevertheless the third technique is the most promising for it is the most efficient in all parameters (see Table 2 for a comparison), the first and second techniques are also beneficial since they work in a higher level of abstraction and assume less about the circuit-garbling functionality. In particular, the first technique can be used with any underlying circuit-based protocol for evaluating the CPU-step circuits. The second technique requires an underlying protocol that relies on the idea of two keys per wire, such as the BMR construction, however it assumes nothing about the way BMR is implemented (recall that BMR on its own uses another MPC protocol to garble the gates). On the contrary, the third technique assumes a specific implementation of BMR, which shares the wires' permutation bits among the parties. The SPDZ protocols family satisfies this last requirement and therefore we use it in our implementation.

2. **Reduced Round and Space Complexities.** As opposed to [22,24] that require communication rounds for every layer of $\mathrm{C}_{\mathrm{CPU}}$, and [29] that achieves only passive security, our protocol is constant round per physical memory access. As mentioned above, the parties can travel from one CPU-step to the next by simply performing SPDZ openings, which appears more efficient than using xor-homomorphic commitment to wire labels in a cut-and-choose based protocol such as [1] (for 2PC).

   We show that by representing memory as a "packed shares" the parties need to store *only 2 bits* per bit in the ORAM (that is, to operate an ORAM with $N'$-bit storage the previous parties need to store $2N'$ bits). To the best of our knowledge this is the best concrete overhead that has been achieved to date. In contrast, other BMR-based protocols, such as one instantiated using our second technique, requires each party to store $n\kappa$ bits per bit in the ORAM. We further devise a way to shrink the storage required by each party in the online phase. When using a garbling scheme that produces a garbled circuit of size independent in the number of parties (as recently proposed [3]) our optimization leads to a decrease in memory consumption of up to 2. We stress that this improvement is applicable to *all* BMR-based constructions. We present and prove security of it in Sect. 6.
3. **Implementation.** We have implemented the protocol using our most efficient memory access technique and obtained experimental access times results in both LAN and simulated WAN environments for two and three participants. In addition, we provide a comparison with the previous implementation of Keller and Scholl [6,24] that is based purely on SPDZ. Our experiments show that [24] performs better over LAN (up to a factor of two for two parties) while our work does so over WAN (by one order of magnitude for two parties), justifying our efforts to reduce communication rounds. This supports the analysis that garbled circuits are more suitable for a setting with high latency because computation on secret values (after obtaining the garbled circuit) can be entirely done locally. Note, however, that we still require communication for revealing memory addresses and transferring memory values to garbled circuit wires. This is not the case for the trivial (asymptotically more expensive) approach where the whole memory is scanned for every access. We also implemented the latter and found that our protocol breaks even at a memory size in the 1'000s for the LAN setting and in the 100'000s for the WAN setting.

We stress that even though [38] also achieves a constant-round multiparty protocol for circuit-based computation (i.e. not RAM programs) our third and most efficient technique is not directly applicable to their construction. In particular, our technique relies on the fact that all parties can identify the correctness of wire labels without communication. This is the case for BMR because every party learns both possibilities for $\kappa$ bits of every wire label. This is only true for one of two parties in the above work. We therefore leave it as an open problem how to combine the two techniques and how a possible combination would compare to our work.

### 1.3 Related Work

Gordon et al. [18] (who followed the work of Ostrovsky and Shoup [33] that was tailored specifically for PIR) designed the first *general* two-party, semi-honest, secure computation protocol for RAM. Their work focuses on the client-server settings, where the client has a small input and the server has a large database, and require the client to maintain only a small storage (i.e. logarithmic in the size of the database). Their technique relies on the one-time-initialization of the ORAM, after which, the server stores an encrypted version of the memory, then the parties iteratively engage in a traditional, circuit-based, secure two-party computation for every ORAM instruction.

Garbled RAM, introduced by Lu and Ostrovsky [30], is an analogue object of garbled circuit with respect to RAM programs. Namely, a user can garble an arbitrary RAM program directly without converting it into a circuit first. A garbled RAM scheme can be used to garble the memory, the program and the input in a way that reveals only the evaluation outcome and nothing else. The main advantage of garbled RAM is that it leads to a constant-round two-party or multi-party protocols to both semi-honest and malicious settings. This is reflected in a series of works on variations of garbled RAM [11–14, 19], however all of these works focused on showing feasibility rather than efficiency and are impractical.

Afshar et al. [1] presented two actively secure protocols for the two-party settings: One that works in the offline-online model and one for streaming. The main idea in both of their schemes is encoding RAM memory via wire labels[4]. When the program reads from memory location $\ell$, it is possible to reuse the appropriate output wire labels from the most recent circuit to write to location $\ell$ (which is not necessarily the previous circuit). Those protocols require the parties to coordinate before the evaluation of each CPU-step, either by soldering techniques that require XOR homomorphic commitments for aligning wire labels (based on [10, 32]) or by invocations of oblivious transfer to allow evaluation of next garbled circuits, in addition to a large amount of symmetric operations for garbling, encrypting and decrypting $s$ copies of the circuit (since it uses the cut-and-choose technique). Overall, this would incur an additional overhead of $O(sn\kappa)$, since for each input wire, of each of the $O(s)$ garbled circuits, each party would need to commit and open its XOR homomorphic commitment, with computational security parameter $\kappa$. Moreover, the streaming version requires both the garbler and the evaluator to maintain $O(s)$ copies of the memory. That work was followed by [20, 31] to achieve a constant round protocol for ZKP of non-algebraic statements in the RAM model, but not for secure computation.

Keller and Scholl [24], showed how to implement two ORAM variants for the oblivious array and oblivious dictionary data structures, specifically, they compared their implementation for the binary Tree ORAM [34] and the Path ORAM

[4] Encoding the *state* as wire labels is simpler than encoding the *memory* since it only requires matching wire labels of output wires of one CPU-step to the input wires of the next. This can be done in the offline phase, without knowing the program or the input.

[35] using various optimizations for many parts of the ORAM algorithms. Their implementation of secure oblivious array and dictionary are purely based on the SPDZ protocol, hence, they have no use of the techniques we develop in this paper because the memory in their work is represented exactly the same as the secret state of the program is represented. Therefore, there is no requirement of conversion between those two entities (memory and state). Due to their use of a secret-sharing based MPC using the SPDZ authenticated shares representation, evaluation of multiplication gates are performed interactively such that the product results are immediately authenticated, thus, parties can use the memory as usual shared secrets and verify authenticity only once, when the evaluation is finished. The drawback in their approach is the high round complexity that is implied on top of the ORAM round complexity. In our protocol, multiplications are evaluated inside a circuit and the authentication of the result is not an integrated part of the multiplication itself (as in the SPDZ protocol).

Doerner and Shelat [9] recently published a *two-party passively* secure computation for RAM programs and reported that it outperforms previous works, even when implemented using the state-of-the-art ORAM schemes, up to large memory sizes such as $2^{32}$ elements of 4 bytes. Their Distributed ORAM scheme (AKA *Floram*) is derived from the Function Secret Sharing (FSS) for point functions by Boyle et al. [4,5], which resembles the trivial ORAM that read/write all memory addresses for every access in order to hide its access pattern, however, this is resolved since those $O(n)$ accesses are performed by a highly parallelizable local computation. The main advantage of Floram is that it has only $O(1)$ communication rounds for both initialization and memory access and does not require secure computation at all for the initialization. We remark that even though FSS is feasible in the multiparty setting, it does not offer the same optimizations as it does to the two party setting, thus, Floram is currently not suitable for the multiparty setting. In addition, it is not trivial to lift their scheme to have active security.

## 2 Preliminaries

Relying on the notation and description of the RAM model of computation presented in Sect. 1.1, we directly proceed to the definition of Oblivious RAM:

### 2.1 Oblivious RAM

A polynomial time algorithm $C$ is an *Oblivious RAM (ORAM)* compiler with computational overhead $c(\cdot)$ and memory overhead $m(\cdot)$, if $C$, when given a security parameter $\kappa$ and a deterministic RAM program $\Pi$ with memory $D$ of size $N$, outputs a program $\Pi'$ with memory $D'$ of size $N' = m(N) \cdot N$, such that for every input $x \in \{0,1\}^*$ the running time of $\Pi'(D', x)$ is bounded by $T' = T \cdot c(N)$ and there is a negligible function $\mu$ such that the following properties hold:

- **Correctness.** For every memory size $N \in \mathbb{N}$ and every input $x \in \{0,1\}^*$ with probability at least $1 - \mu(\kappa)$, the output of the compiled program equals the output of the original program, i.e. $\Pi'(D', x) = \Pi(D, x)$.
- **Obliviousness.** For every two programs $\Pi_1, \Pi_2$, every $D_1, D_2$ of size $N$ and every two inputs $x_1, x_2 \in \{0,1\}^*$, if the running times of $\Pi_1(D_1, x_1)$ and $\Pi_2(D_2, x_2)$ are $T$, then

$$\mathsf{AP}(C(\Pi_1, \kappa), D_1, x_1)) \overset{c}{\equiv} \mathsf{AP}(C(\Pi_2, \kappa), D_2, x_2))$$

where $\mathsf{AP}(\cdot)$ is the access pattern as defined in Sect. 1.1.

As reflected from the above definition, our ORAM scheme is required to hide only the *addresses* that $\mathsf{CPU}$ accesses since we handle the privacy and authenticity of the contents of the memory using other techniques. Also, note that the definition does not require to hide the runtime of the program.

### 2.2 Secure Computation in the RAM Model

Informally, a secure protocol for RAM programs must hide both program's access pattern and its memory contents from the parties. In addition, it must keep the memory "fresh", that is, it prevents the adversary to plug in an outdated memory block to the current CPU-step circuit.

Protocols in this model [13,14,19] typically induce two flavors of security definitions, such that their construction could be modular, i.e. first achieve a construction for the weaker security notion (usually called Unprotected Memory Access) and then enhance it with an ORAM to achieve full security. Informally, the definition of full security requires that the access pattern remains hidden, that is, the ideal adversary only obtains the runtime $T$ of the program $\Pi$ and the computation output $\boldsymbol{y}$. Given only $T$ and $\boldsymbol{y}$, the simulator must be able to produce an indistinguishable access pattern. The weaker notion of security, as known as Unprotected Memory Access (UMA), leaks the memory contents as well as the access pattern to the adversary. In fact, UMA-secure protocols only deal with how to authentically pass a memory block written in the past to a circuit that needs to read it in a later point in time. In this work we use the same definition for full security, however, we use a different definition, called Unprotected Access Pattern (UAP) instead of the UMA. The definition of UAP is stronger than UMA since it requires the memory contents remain hidden from the adversary (and only the access pattern is leaked). Recall that since our construction is for the non-oblivious computation (see Sect. 1.1) in both security notions the adversary receives the instruction pattern as well.

Obviously, using a standard ORAM scheme we can easily transform a protocol that is UAP secure to a protocol that is fully secure [19], therefore, we may focus on the weaker notion (although our implementation achieves full security). We proceed to define both notions.

**Full Security.** Following the simulation paradigm [15, Chap. 7] we present the ideal and real models of executions of RAM programs.

**Functionality $\mathcal{F}_{\mathrm{RAM}}$**

The functionality interacts with parties $p_1, \ldots, p_n$ and the adversary $\mathcal{A}$. The program $\Pi$ is known and agreed by all parties. The functionality initializes memory $D$ of size $N$ blocks.
**Input.** Party $p_i$ has its private input $x^i$.
**Output.** Execute $\boldsymbol{y} \leftarrow \Pi(D, \boldsymbol{x})$ and send $(T, \boldsymbol{y})$ and $\mathsf{IP}(\Pi, D, \boldsymbol{x})$ to $\mathcal{A}$ (where $T$ is the runtime of the execution). If $\mathcal{A}$ returns $\mathsf{Abort}$ then halt, otherwise output $(T, \boldsymbol{y})$ and $\mathsf{IP}(\Pi, D, \boldsymbol{x})$ to all honest parties.

**Fig. 1.** Ideal execution of $\Pi(N, \boldsymbol{x})$ with abort.

*Execution in the ideal model.* In an ideal execution $\mathcal{F}_{\mathrm{RAM}}$ (Fig. 1), the parties submit their inputs to a trusted party which in turn executes the program and returns the output. Let $\Pi$ be a program with memory $D$ of size $N$, which expects $n$ inputs $\boldsymbol{x} = x^1, \ldots, x^n$, let $\mathcal{A}$ be a non-uniform PPT adversary and let $I \subset [n]$ be the set of indices of parties that $\mathcal{A}$ corrupts; we may refer to the set of corrupted parties by $p_I$. Denote the *ideal execution of* $\Pi$ on $\boldsymbol{x}$, auxiliary input $z$ to $\mathcal{A}$ and security parameter $\kappa$ by the random variable $\mathbf{IDEAL}^{\mathcal{F}_{\mathrm{RAM}}}_{\mathcal{A}(z),I}(\kappa, \Pi, D, \boldsymbol{x})$, as the output set of the honest parties and the adversary $\mathcal{A}$.

*Execution in the real model.* In the real model there is no trusted party and the parties interact directly. The adversary $\mathcal{A}$ sends all messages in place of the corrupted parties, and may follow an arbitrary PPT strategy whereas honest parties follow the protocol. Let $\Pi, D, \mathcal{A}, I$ be as above and let $\mathcal{P}$ be a multiparty protocol for computing $\Pi$. The *real execution of* $\Pi$ on input $\boldsymbol{x}$, auxiliary input $z$ to $\mathcal{A}$ and security parameter $\kappa$, denoted by the random variable $\mathbf{REAL}^{\mathcal{P}}_{\mathcal{A}(z),I}(\kappa, \Pi, D, \boldsymbol{x})$, is defined as the outputs set of the honest parties and the adversary $\mathcal{A}$.

**Definition 2.1 (Secure computation).** *Protocol $\mathcal{P}$ is said to* securely compute $\Pi$ with abort in the presence of malicious adversary *if for every PPT adversary $\mathcal{A}$ in the real model, there exists a PPT adversary $\mathcal{S}$ in the ideal model, such that for every $I \in [n]$, every $\boldsymbol{x}, z \in \{0,1\}^*$ and for large enough $\kappa$, the following holds*

$$\left\{\mathbf{IDEAL}^{\mathcal{F}_{\mathrm{RAM}}}_{\mathcal{S}(z),I}(\kappa, \Pi, D, \boldsymbol{x})\right\}_{\kappa,\boldsymbol{x},z} \overset{c}{\equiv} \left\{\mathbf{REAL}^{\mathcal{P}}_{\mathcal{A}(z),I}(\kappa, \Pi, D, \boldsymbol{x})\right\}_{\kappa,\boldsymbol{x},z}$$

**Unprotected Access Pattern (UAP) Security.** This notion allows the adversary to further inspect the access pattern. The ideal functionality $\mathcal{F}_{\mathsf{UAP}}$ is given in Fig. 2 and realized by protocol $\mathcal{P}_{\mathsf{UAP}}$ (Fig. 7).

**Definition 2.2 (Secure computation in the UAP model).** *Protocol $\mathcal{P}$ is said to* securely compute $\Pi$ in the UAP model with abort in the presence of malicious adversary *if for every PPT adversary $\mathcal{A}$ for the real model, there*

**Functionality $\mathcal{F}_{\mathsf{UAP}}$**

The functionality interacts with parties $p_1, \dots, p_n$ and the adversary $\mathcal{A}$. The program $\mathit{\Pi}$ is known and agreed by all parties. The functionality initializes memory $D$ of size $N$ blocks.

**Input.** Party $p_i$ has its private input $x^i$.

**Output.** Execute $\boldsymbol{y} \leftarrow \mathit{\Pi}(D, \boldsymbol{x})$ and send $\mathsf{AP}(\mathit{\Pi}, D, \boldsymbol{x})$, $\mathsf{IP}(\mathit{\Pi}, D, \boldsymbol{x})$ and $\boldsymbol{y}$ to $\mathcal{A}$. If $\mathcal{A}$ returns $\mathsf{Abort}$ then halt, otherwise output $\mathsf{AP}(\mathit{\Pi}, D, \boldsymbol{x})$, $\mathsf{IP}(\mathit{\Pi}, D, \boldsymbol{x})$ and $\boldsymbol{y}$ to all honest parties.

**Fig. 2.** Ideal execution of $\mathit{\Pi}(N, x)$ in the UAP model.

*is a PPT adversary $\mathcal{S}$ for the ideal model, such that for every $I \in [n]$, every $\boldsymbol{x}, z \in \{0,1\}^*$ and for large enough $\kappa$*

$$\left\{\mathbf{IDEAL}^{\mathcal{F}_{\mathsf{UAP}}}_{\mathcal{S}(z),I}(\kappa, \mathit{\Pi}, D, \boldsymbol{x})\right\}_{\kappa,\boldsymbol{x},z} \overset{c}{\equiv} \left\{\mathbf{REAL}^{\mathcal{P}}_{\mathcal{A}(z),I}(\kappa, \mathit{\Pi}, D, \boldsymbol{x})\right\}_{\kappa,\boldsymbol{x},z}$$

The transformation (or compilation) from UAP to full security is not in the scope of this paper and can be found in previous works [11–14,19]. We follow that path since it makes the security analysis simpler and modular, rather than proving full security from scratch. Therefore, functionality $\mathcal{F}_{\mathsf{UAP}}$ in Fig. 2, which is realized in protocol $\mathcal{P}_{\mathsf{UAP}}$ (Fig. 7), reveals the access pattern to the parties. By incorporating an ORAM scheme on top of our protocol that access pattern would be of no gain to the adversary for the reason that an access pattern of a program execution using an ORAM is indistinguishable from an access pattern of a randomly chosen program with the same runtime.

We note that achieving a UAP-secure protocol may be useful on its own (i.e. without lifting it up to full security) in cases where the original program $\mathit{\Pi}$ is oblivious, that is, when the access pattern is *permitted* to be leaked to the parties.

# 3 Executing RAM Programs Using BMR

Our protocol follows the BMR-SPDZ approach [25,27] and adapts the free-XOR technique for the BMR garbling scheme [2]. For completeness, in the following we describe the structure of the actively secure additive secret sharing used in SPDZ-like protocols and outline the BMR-SPDZ approach.

## 3.1 SPDZ Secret Sharing

SPDZ-like protocols use actively secure additive secret sharing over a finite field, combined with information theoretic MACs to ensure active security. A shared secret $x \in \mathbb{F}$ is represented by

$$[\![x]\!] = ([x], [m(x)], [\alpha]) = (x^1, \dots, x^n, m(x)^1, \dots, m(x)^n, \alpha^1, \dots, \alpha^n)$$

where $m(x) = x \cdot \alpha$ is a MAC on message $x$ using a global key $\alpha$. Party $p_i$ holds: A uniformly random share $x^i$ of $x$, a uniformly random share $m(x)^i$ of $m(x)$ and a uniformly random share $\alpha^i$ of $\alpha$ such that

$$x = \sum_{i=1}^{n} x^i, \qquad m(x) = \sum_{i=1}^{n} m(x)^i, \qquad \alpha = \sum_{i=1}^{n} \alpha^i$$

We denote an additive secret shared value $x$ by $[x]$ and its authenticated shared version by $[\![x]\!]$. We also denote $p_i$'s share by $[\![x]\!]^i = (x^i, m(x)^i)$.

When opening a shared value $[\![x]\!]$ the parties first broadcast their shares $x^i$ and compute $x$. To ensure that $x$ is correct, they then check the MAC by committing to and opening $m(x)^i - x \cdot \alpha^i$ and checking these shares sum up to zero.

### 3.2 The BMR-SPDZ Protocol

Unlike the two-party settings, in which we have one garbler and one evaluator, in the multiparty settings *all* parties are both garblers and evaluators such that no strict subset of parties can either influence or learn anything about the values that the wires carry. In the following we present the key points in the BMR-SPDZ approach:

*Keys.* Every party chooses a random key for each wire in the circuit, that is, party $p_i$ chooses key $k^i_w \in \mathbb{F}_{2^\kappa}$ for wire $w$. This key is named "0-key" and denoted $k^i_{w,0}$ where $k^i_{w,0}$ is essentially the $i$-th coordinate of a *full* 0-key, $k_{w,0} = (k^1_{w,0}, \ldots, k^n_{w,0}) \in (\mathbb{F}_{2^\kappa})^n$.

*Global difference.* To enable free-XOR, each party chooses its own global-difference, that is, party $p_i$ randomly chooses $\varDelta_i$ such that the difference between its 0-key and its 1-key is $\varDelta_i$. Formally, $k^i_{w,1} = k^i_{w,0} \oplus \varDelta_i$ for every $w$ and $i$. Similarly $\varDelta_i$ is the $i$-th coordinate of the full difference $\varDelta = (\varDelta_1, \ldots, \varDelta_n)$. The value $\varDelta_i$ is known only to party $p_i$ and no strict subset of the parties (that does not include $p_i$) can learn it. For wire $w$ we get that $k_{w,1} = k_{w,0} \oplus \varDelta$ where $\oplus$ operates component-wise.

*Permutation bits.* In the course of the evaluation the parties obtain $k_{w,b}$ with either $b = 0$ or $b = 1$ for every wire $w$. Party $p_i$ could easily check whether $b = 0$ or $b = 1$ by extracting the $i$th element from $k_{w,b}$ and compare it to $k^i_{w,0}$ and $k^i_{w,1}$. If $b = 0$ we say that the *external value* of wire $w$, denoted $\varLambda_w$ is 0, otherwise, if $b = 1$, then $\varLambda_w = 1$. Since the *real value* that is carried by wire $w$, denoted by $\rho_w$, must be kept secret, the external value $\varLambda_w$ must reveal nothing about it. To this end, a random *permutation bit*, $\lambda_w$, is assigned to each wire $w$ in order to mask $\rho_w$ by setting $\varLambda_w = \lambda_w \oplus \rho_w$.

*Inputs.* Let $w$ be an input wire that is associated with input $x^i$ of party $p_i$, then the parties open $\lambda_w$ to party $p_i$ only. Then $p_i$ broadcasts $\varLambda_w$ and $k^i_{w,\varLambda_w}$ where

$\varLambda_w = \rho_w \oplus \lambda_w$ and $\rho_w$ is its input to wire $w$. Then, party $p_j$, for all $j$, broadcasts its $\varLambda_w$-key $k^j_{w,\varLambda_w}$ such that all parties obtain $k_{w,\varLambda_w} = (k^1_{w,\varLambda_w}, \ldots, k^n_{w,\varLambda_w})$.

*Outputs.* If $w$ is an output wire then the parties open the permutation bit $\lambda_w$ to everyone. This way, upon obtaining key $k_{w,\varLambda_w}$ the parties learn the real value of $w$ by $\rho_w = \varLambda_w \oplus \lambda_w$.

*Encrypting a key.* In the process of garbling, the parties encrypt the key of a gate's output wire using the keys of its input wires. Let $m = m^1, \ldots, m^n$ be the key to be encrypted and $k_\ell, k_r$ with $k_b = k^1_b, \ldots, k^n_b$ be encryption keys of the left and right input wires, where party $p_i$ has $m^i, k^i_\ell, k^i_r$. The parties produce the ciphertext $c = c^1, \ldots, c^n$ as follows: $m^j$ is encrypted using $k_\ell, k_r$ to result $c^j$ such that even a single missing coordinate of $k_\ell$ and $k_r$ prevents one from decrypting $c^j$. To encrypt $m^j$, party $p_i$ provides $F_{k^i_\ell}(j), F_{k^i_r}(j)$, where $F$ is a pseudorandom generator and then, using a protocol for secure computation the parties evaluate and output:

$$c^j = \mathsf{Enc}_{k_\ell,k_r}(m^j) = \left(\bigoplus_{i=1}^{n} F_{k^i_\ell}(j)\right) \oplus \left(\bigoplus_{i=1}^{n} F_{k^i_r}(j)\right) \oplus m^j$$

Note that the keys $k^i_\ell, k^i_r$ are necessary for the decryption of $c^j$ for every $i, j \in [n]$.

*Garbled gate.* A garbled version of an AND gate $g$ with input wires $u, v$ and output wire $w$, is simply a 4-entries table, each entry is an encryption of either $k_{w,0}$ or $k_{w,1}$, this depends on the permutation bits $\lambda_u, \lambda_v$ and $\lambda_w$. We want to enable the evaluator, who holds $k_{u,\varLambda_u}$ and $k_{v,\varLambda_v}$ (which are translated to $\rho_u$ and $\rho_v$ respectively) to decrypt the ciphertext in the $(2\varLambda_u + \varLambda_v)$-th entry of the table and obtain $k_{w,\varLambda_w}$ such that $\rho_w = \rho_u \cdot \rho_v$. That is, we want to have $\lambda_w \oplus \varLambda_w = (\lambda_u \oplus \varLambda_u) \cdot (\lambda_v \oplus \varLambda_v)$, thus, the $(2\varLambda_u + \varLambda_v)$-th entry conceals $k_{w,\varLambda_w}$ where

$$\varLambda_w = (\lambda_u \oplus \varLambda_u) \cdot (\lambda_v \oplus \varLambda_v) \oplus \lambda_w$$

and since $k_{w,1} = k_{w,0} \oplus \varDelta$ we get that the entry conceals

$$k_{w,0} + \varLambda_w \cdot \varDelta = k_{w,0} + \big((\lambda_u \oplus \varLambda_u) \cdot (\lambda_v \oplus \varLambda_v) \oplus \lambda_w\big) \cdot \varDelta$$

We conclude by presenting functionality $\mathcal{F}_{\mathsf{BMR}}$ (Fig. 3) for a construction of a garbled circuit. Note that the only difference between $\mathcal{F}_{\mathsf{BMR}}$ to the standard description of this functionality [2,25] is that here the functionality lets the parties learn a share to the permutation bits $\lambda_w$. This is necessary in order to obtain a neat security proof of the construction. Protocol $\mathcal{P}_{\mathsf{BMR}}$ (Fig. 5) realizes $\mathcal{F}_{\mathsf{BMR}}$ in the $\mathcal{F}_{\mathsf{MPC}}$-hybrid model (Fig. 11 in the appendix). Given a garbled circuit the parties evaluate it using the $\mathcal{E}_{\mathsf{BMR}}$ procedure described in Fig. 4.

In the presentation of the protocol (Fig. 5) and to the rest of the paper, we denote by $\langle x \rangle$ the handler (*varid*) of a variable $x$ that is stored by $\mathcal{F}_{\mathsf{MPC}}$.

**Functionality $\mathcal{F}_{\mathsf{BMR}}$**

The functionality interacts with $n$ parties $p_1, \ldots, p_n$ and exports the following instructions:

**Garble** On input $(\mathsf{Garble}, C, (W, P))$ from all parties where $C$ is the circuit to garble and $(W, P)$ is a map from a wire to a party (used to associate an input wire to the party who feeds its input to it, i.e. $W[\ell]$ is an input wire associated with party $P[\ell]$).

1. For party $P_i$:
   (a) Sample and output to $p_i$ a global random difference $\Delta_i \in \mathbb{F}_{2^\kappa}$.
   (b) Sample a key $k^i_{w,0} \in \mathbb{F}_{2^\kappa}$ for all $w \in C$ (such that $k^i_{w,1} = k_{w,0} \oplus \Delta_i$).
   (c) Output $k^i_{w,0}$ to $P_i$ for all input wires in $C$.
2. Samples a permutation bit $\lambda_w \in \{0, 1\}$ for all $w \in C$.
3. Create SPDZ share of $\lambda_w$ for every memory access (reading and writing) wire $w \in C$ and send $[\![\lambda_w]\!]^i$ to $p_i$.
4. For every $1 \leq \ell \leq |W|$: Output $\lambda_{W[\ell]}$ in the clear to $P[\ell]$.
5. For every AND gate $g \in C$ with input wires $u, v$ and output wire $w$, every $\Lambda_u, \Lambda_v \in \{0, 1\}$, and every $j \in [n]$, compute:

$$\tilde{g}^j_{\alpha,\beta} = \left( \bigoplus_{i=1}^{n} F_{k^i_{u,\Lambda_u}}(g, j) \bigoplus_{i=1}^{n} F_{k^i_{v,\Lambda_v}}(g, j) \right) \oplus k^j_{w,0} \quad (2)$$
$$\oplus \Delta_j \cdot \big( (\lambda_u \oplus \alpha) \cdot (\lambda_v \oplus \beta) \oplus \lambda_w \big)$$

and store $GC = \big\{ \{ (\tilde{g}^1_{\Lambda_u,\Lambda_v}, \ldots, \tilde{g}^n_{\Lambda_u,\Lambda_v}) \}_{\Lambda_u,\Lambda_v \in \{0,1\}} \big\}_{g \in G}$

**Open** Output $GC$ to all parties.

**Fig. 3.** The BMR functionality.

### 3.3 Towards RAM Computation

To be able to securely compute RAM programs (in the UAP model) the parties garble $T$ circuits $GC^1, \ldots, GC^T$ and then evaluate them sequentially. To this end, we must specify how the parties obtain the keys intended for the input wires of each garbled circuit (these are the input wires associated with values $\mathsf{state}_t$ and $b^{\mathsf{read}}_t$). This task is divided in two: First, the input wires of $GC^t$ associated with $\mathsf{state}_t$ must carry the same values as the output wires associated with $\mathsf{state}_t$ in $GC^{t-1}$. Second, we need to support secure memory access, that is, the input wires of $GC^t$ associated with $b^{\mathsf{read}}_t$ must carry the same values as the output wires associated with $b^{\mathsf{write}}_{t'}$ in $GC^{t'}$, where $t'$ is the most recent timestep in which address $i^{\mathsf{read}}_{t-1}$ was modified. The first task could be easily achieved by changing $\mathcal{P}_{\mathsf{BMR}}$ to choose the same keys for both output and input wires that are associated to the same state in every two consecutive garbled circuits, however, this would be non-black-box in $\mathcal{F}_{\mathsf{BMR}}$ (since the functionality chooses its keys independently for every circuit). For a black-box solution we can use the techniques described in Sects. 4.2 and 4.3. We stress, though, that the two tasks are orthogonal and

**Procedure $\mathcal{E}_{\mathsf{BMR}}$**

**Evaluate** On input $(\mathsf{Eval}, K)$ from all parties, where $K = \{k_w\}_w$ for all input wires in $C$, evaluate $GC$ as follows: Traverse $C$ in a topological order. Let $g$ be the current gate with input wires $u, v$ and output wire $w$ and $k_{u,\Lambda_u}, k_{v,\Lambda_v}$ the keys obtained for wires $u$ and $v$ respectively. Then,

1. If $g$ is a XOR gate, compute the output key and its external value by $k_{w,\Lambda_w} = k_{u,\Lambda_u} \oplus k_{v,\Lambda_v}$ and $\Lambda_w = \Lambda_u \oplus \Lambda_v$ where $\oplus$ operates component-wise.
2. If $g$ is an AND gate, compute
$$k^j_{w,\gamma} = \tilde{g}^j_{\alpha,\beta} \oplus \left( \bigoplus_{i=1}^{n} F_{k^i_{u,\alpha}}(g,j) \bigoplus_{i=1}^{n} F_{k^i_{v,\beta}}(g,j) \right)$$
for every $j \in [n]$ where $\alpha = \Lambda_u$ and $\beta = \Lambda_v$. Set $k_{w,\gamma} = (k^1_{w,\gamma}, \dots, k^n_{w,\gamma})$. For party $p_i$: If $k^i_{w,\gamma} = k^i_{w,0}$ set $\Lambda_w = 0$, otherwise, if $k^i_{w,\gamma} = k^i_{w,1}$ set $\Lambda_w = 1$, otherwise abort.

**Fig. 4.** Evaluation of a BMR garbled circuit.

**Protocol $\mathcal{P}_{\mathsf{BMR}}$**

The parties initialize $\mathcal{F}_{\mathsf{MPC}}$ by calling $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Init}(k)$.

**Garble**

1. For each $i \in [n]$ invoke $\langle \Delta_i \rangle \leftarrow \mathcal{F}_{\mathsf{MPC}}.\mathsf{Random}()$. This is party $p_i$'s global difference. Then call $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Output}(\langle \Delta_i \rangle, i)$.
2. Output $k^i_{w,0}, k^i_{w,1}$ to $p_i$ using $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Output}$ where $\langle k^i_{w,0} \rangle \leftarrow \mathcal{F}_{\mathsf{MPC}}.\mathsf{Random}()$ and $\langle k^i_{w,1} \rangle \leftarrow \mathcal{F}_{\mathsf{MPC}}.\mathsf{Add}(\langle k^0_{w,0} \rangle, \langle \Delta_i \rangle)$ for all $i \in [n]$ and $w \in W$.
3. For every wire $w \in W$ invoke $\langle \lambda_w \rangle \leftarrow \mathcal{F}_{\mathsf{MPC}}.\mathsf{RandomBit}()$.
4. Party $p_i$ calls $\langle F_{k^i_{w,b}}(g,j) \rangle \leftarrow \mathcal{F}_{\mathsf{MPC}}.\mathsf{Input}(F_{k^i_{w,b}}(g,j))$ for every gate $g \in G, j \in [n]$ and $b \in \{0,1\}$ where $w$ is either the first or the second input wire for gate $g$.
5. Compute $\langle GC \rangle = \left\{\{[\![\tilde{g}^j_{\alpha,\beta}]\!]\}_{j\in[n],\alpha,\beta\in\{0,1\}}\right\}_{g\in G}$ with $\tilde{g}^j_{\alpha,\beta}$ as in Equation 2, where additions are computed using $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Add}$ and multiplications are computed using $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Multiply}$.

**Open** Invoke $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Open}$ on $\tilde{g}^j_{\alpha,\beta}$ for every $j \in [n], \alpha, \beta \in \{0,1\}$ and $g \in G$.

**Fig. 5.** Realizing $\mathcal{F}_{\mathsf{BMR}}$ in the $\mathcal{F}_{\mathsf{MPC}}$-hybrid model.

the techniques chosen to complete them are independent. Therefore, in the rest of the presentation we focus on realizing secure memory access (the second task) while taking for granted the traveling of the CPU's state (i.e. we may write *"the parties obtain the input wires of* $\mathsf{state}_t$" without specifying how).

# 4 Accessing Memory

In this section we present the three techniques to achieve secure memory accesses and show how to realize $\mathcal{F}_{\mathsf{UAP}}$ in the $\mathcal{F}_{\mathsf{BMR}}$-hybrid model using the third one. We compare the performance of the techniques in Table 2. The values within the table are explained alongside the description of the techniques.

In the presentation below, we group some set of input/output wires together, according to their purpose as follows: $W_{\mathsf{in}}$ refers to the input wires of $GC^1$, which correspond to the parties' inputs where $W^i_{\mathsf{in},j}$ corresponds to the $j$-th bit of input $x^i$. $W^t_{\mathsf{read}}$, $W^t_{\mathsf{write}}$,$W^t_{\mathsf{addr,rd}}$, $W^t_{\mathsf{addr,wr}}$ refer to the input $b^{\mathsf{read}}_t$, output $b^{\mathsf{write}}_t$, addresses $i^{\mathsf{read}}_t, i^{\mathsf{write}}_t$ respectively in $GC^t$. In addition, $W^t_{\mathsf{state}}$ refers to the input state and $W^t_{\mathsf{state'}}$ refers to the output state in $GC^t$.

## 4.1 Memory via Embedded Authentication Sub-circuit

This technique assumes that the values $b^{\mathsf{read}}_t, b^{\mathsf{write}}_t$ are elements from the same field that SPDZ use as the underlying MPC protocol. This way, if SPDZ statistical parameter is $s$ then the memory is divided into data items of $s$ bits. We enhance $\mathrm{C_{CPU}}$ with two procedures: Verify and AuthShare and denote the result

**Table 2.** Performance of the three techniques with $n$ parties. $\kappa$ and $s$ are the computational and statistical security parameters. The columns specify the following parameters: *Number of input wires* required in the CPU-step circuit for every input bit (of the ORAM) that is being read by that circuit. *Amount of communication* required for each memory access, this is measured in bits per input wire per party. *Number of communication rounds* required from the moment the parties obtained keys of output wires of $GC^t$ to the moment they obtain keys to the input wires of $GC^{t+1}$. Communication rounds could be used for secret opening, broadcasting a value or performing multiplication over shares; among the three, only multiplication requires more work to be done in the offline phase, specifically, multiplication requires sacrificing a multiplication triple. This is reflected in the Triples column, note that we multiply a *vector* of $n$ keys rather than a single key. *Memory overhead* specifies how many bits do we store in the memory for a single bit of ORAM memory (again, this is per party. The total memory size that the parties store should be multiplied by $n$). The last column specifies whether a change in the garbled circuit is needed.

| | Input wires | Communication | Rounds | Triples | Memory overhead | Requires change in $\mathrm{C_{CPU}}$ |
|---|---|---|---|---|---|---|
| Embedded subcircuit (Sect. 4.1) | $4n$ | $4\kappa$ | 2 | $O(ns^2)$ | 2 | $+2s^2$ AND gates $+4sn$ input wires per data item |
| Soldering (Sect. 4.2) | 1 | $(3n+1)\kappa$ | 2 | $2n$ | $n\kappa$ | No |
| **Shared bits** (Sect. 4.3) | 1 | $\mathbf{2}s$ | **2** | **0** | **2** | **No** |

**Enhanced CPU-step circuit** $\mathrm{C}_{\mathrm{CPU}^+}$

**Input.** The parties input $\mathsf{state}$ and $[\![b^{\mathsf{read}}]\!]$. In addition they input 2 random values $[r_1], [r_2]$. That is, party $p_i$ inputs $(b^{\mathsf{read}})^i, m(b^{\mathsf{read}})^i, \alpha^i, r_1^i, r_2^i$.

**Output.**

1. Compute $\mathsf{ver} = \mathsf{Verify}([\![b^{\mathsf{read}}]\!])$.
2. Compute $(\mathsf{state}', i^{\mathsf{read}}, i^{\mathsf{write}}, b^{\mathsf{write}}) = \mathrm{C}_{\mathrm{CPU}}(\mathsf{state}, b^{\mathsf{read}})$
3. Compute $(\mathsf{val}, \mathsf{mac}) = \mathsf{AuthShare}(b^{\mathsf{write}}, [\alpha], [r_1], [r_2])$.
4. Output $\mathsf{state}', \mathsf{ver}, \mathsf{val}$ and $\mathsf{mac}$.

**Fig. 6.** Enhanced CPU-step circuit $\mathrm{C}_{\mathrm{CPU}^+}$

by $\mathrm{C}_{\mathrm{CPU}^+}$ (see Fig. 6). Note that each party has $4s$ input wires for the purpose of authentication and sharing (assuming that the global MAC key is part of the state).

**Privacy.** The parties maintain their memory in the form of SPDZ shares, thus, to input content from location $i^{\mathsf{read}}$ in memory, every party inputs its SPDZ share of this content from its own storage $D[i^{\mathsf{read}}]$. Then the secret is being constructed within the CPU-step circuit. Since this is an additive secret sharing scheme, the content is being constructed using only XOR gates, which requires no communication.

**Verify Authenticity.** We enhance the CPU-step with a sub-circuit that verifies the authenticity of the secret $b^{\mathsf{read}}$, the sub-circuit is denoted $\mathsf{Verify}([\![v]\!])$ where $v$ refers to $b^{\mathsf{read}}$. Party $p_i$ inputs $(v^i, m(v)^i, \alpha^i)$ and the sub-circuit computes[5]:

$$v = \sum_{i=1}^{n} v^i, \qquad m(v) = \sum_{i=1}^{n} m(v)^i, \qquad \alpha = \sum_{i=1}^{n} \alpha^i$$

and outputs $\mathsf{ver} = 1$ if $m(v) - (\alpha \cdot v) = 0$ (meaning verification succeeded) and 0 otherwise (meaning verification has not succeeded), which incurs $s^2$ AND gates for a single multiplication operation in addition to $2s - 1$ AND gates for deciding whether $\mathsf{ver}$ is 1 or 0. Note that this multiplication is over a polynomial ring over $\mathbb{F}_2$, thus the addition involves only XOR. Furthermore, we check the result directly for zero and skip the reduction modulo an irreducible polynomial (mapping from $\mathbb{F}_2[X]$ to $\mathbb{F}_{2^s}$), hence the $2s - 1$ AND gates for comparison.

*Security.* Obviously, since nothing is revealed except from the fact of the authenticity being correct, the adversary cannot extract any information regarding the value.

[5] Remember that $x^i$ denotes the share of party $p_i$ and not an exponentiation operation.

**Authenticated Share.** The CPU-step produces the value $b^{\mathsf{write}}$ to be written to the memory, which obviously could not be output in the clear, rather, it is shared between all parties. The sub-circuit $\mathsf{AuthShare}(v, [\alpha], [r_1], [r_2])$ is given the value $v$ to share (which refers to the value $b^{\mathsf{write}}$), the global MAC key $\alpha$ and two freshly chosen $r_1, r_2 \in \mathbb{F}_{2^s}$ from the parties such that party $p_i$ inputs $(\alpha^i, r_1^i, r_2^i)$. The circuit computes

$$r_1 = \sum_{i=1}^{n} r_1^i, \qquad r_2 = \sum_{i=1}^{n} r_2^i, \qquad \alpha = \sum_{i=1}^{n} \alpha^i$$

and outputs $\mathsf{val} = (v + r_1)$ and $\mathsf{mac} = (\alpha \cdot v + r_2)$. To obtain the SPDZ sharing $[\![v]\!]$, party $p_1$ stores $v^1 = (\mathsf{val} - r_1^1)$ and $m(v)^1 = (\mathsf{mac} - r_2^1)$ and all other parties $p_j$ store $v^j = (-r_1^j)$ and $m(v)^j = (-r_2^j)$.

*Security.* First note that $\sum_{i=1}^{n} v^i = \mathsf{val} - r_1^1 - \sum_{j=2}^{n} r_1^j = v$ and $\sum_{i=1}^{n} m(v)^i = \mathsf{mac} - r_2^1 - \sum_{j=2}^{n} r_2^j = m(v) = \alpha \cdot v$ as required. The values $v$ and its authentication $m(v)$ are independently masked using a truly random values and thus are hidden from any strict subset of parties.

*Performance.* For every data item in $\mathbb{F}_{2^s}$ the parties store its MAC as well, which leads to an overhead of 2 in the memory size. To obtain the key of an input wire of the next circuit each party needs to broadcast its BMR key, which is of size $\kappa$. Since for every read of data item each party inputs 4 $\mathbb{F}_{2^s}$ elements (assuming that the global key $\alpha$ is part of the state), the communication complexity is $4\kappa n$ bits per input bit. The additional circuitry for authentication (and verification) is of size $2s^2 + 2s - 1$ (for two multiplications of elements from $\mathbb{F}_{2^s}$ and additional zero testing), note in Table 2 the required number of multiplications triples is multiplied by $n$ since each AND gate manipulates keys vectors of $n$ coordinates. To obtain the keys for the next CPU-step circuit 2 communication rounds are required, one to broadcast the external value of the input wire (which is done by the party whose input is associated with) and the other is to broadcast the appropriate keys by all parties. Note that we cannot save a communication round by broadcasting the external value when writing (rather than when reading) since the external value that the parties broadcast depends on the input wire of the circuit that is going to read it in a later, unknown point in time.

### 4.2 Memory via Wire Soldering

**The General Technique.** Wire soldering allows the parties to reuse an output wire key of one gate as an input wire key of another gate even if these two gates were not meant to be connected in garbling time. The notion of wire soldering for secure computation was introduced in [32] for the two-party settings and implemented using an additively homomorphic commitment scheme $(\mathsf{com}, \mathsf{dec})$, that is, $\mathsf{com}(a) + \mathsf{com}(b) = \mathsf{com}(a + b)$. Let $u$ and $v$ be wires with keys $k_{u,0}, k_{u,1}, k_{v,0}, k_{v,1}$ and permutation bits $\lambda_u, \lambda_v$. By soldering $v$ to $u$ we would like to achieve the following feature: Obtaining key $k_{u,\Lambda_u}$ that carries a real value $\rho_u = \lambda_u \oplus \Lambda_u$, enables to obtain the key $k_{v,\Lambda_v}$, which carries the *same real value* $\rho_v = \lambda_v \oplus \Lambda_v = \rho_u$. It follows that if $\lambda_u = \lambda_v$ the soldering information reveals $k_{v,\Lambda_u}$ (i.e. $\Lambda_u = \Lambda_v$), otherwise, if $\Lambda_u \neq \Lambda_v$, it reveals $k_{v,(1-\Lambda_u)}$.

In a circuit-based 2PC (with garbler and evaluator) this is done by having the garbler send the commitments $\mathsf{com}(k_{u,0})$, $\mathsf{com}(k_{u,1})$, $\mathsf{com}(k_{v,0})$, $\mathsf{com}(k_{v,1})$; if $\lambda_u = \lambda_v$ the garbler also sends the decommitments $s_0 = \mathsf{dec}(k_{u,0} \oplus k_{v,0})$ and $s_1 = \mathsf{dec}(k_{u,1} \oplus k_{v,1})$, otherwise (if $\lambda_u \neq \lambda_v$) the garbler sends $s_0 = \mathsf{dec}(k_{u,0} \oplus k_{v,1})$ and $s_1 = \mathsf{dec}(k_{u,1} \oplus k_{v,0})$. Given the key $k_{u,\Lambda_u}$, the evaluator computes $k_{u,\Lambda_u} \oplus s_{\Lambda_u}$ to obtain the correct key for wire $k_{v,\Lambda_v}$. To prove that the garbler hasn't inverted the truth value of the wires by choosing the wrong case above, it must also decommit to the XOR of the permutation bits $(\lambda_u \oplus \lambda_v)$. Note that the evaluator learns whether $\Lambda_u = \Lambda_v$ and thus also learns whether $\lambda_u = \lambda_v$, however, that doesn't reveal anything about the real value $\rho_u = \rho_v$ that is carried over those wires.

**Soldering in Our Scheme.** When $\mathcal{P}_{\mathsf{BMR}}$ (Fig. 5) uses SPDZ to garble the circuit party $p_i$ not only obtains its own keys $k^i_{w,b}$ in the clear, but also obtains a SPDZ sharing for both a *whole* keys $k_{w,b}$ and the permutation bits $\lambda_w$ for every $w \in W$ and $b \in \{0,1\}$, thus, the parties could use $\mathcal{F}_{\mathsf{MPC}}$ to perform arithmetic operations over them.

Let $u$ and $v$ be wires with keys $k_{u,0}, k_{u,1} = k_{u,0} \oplus \Delta^u$ and $k_{v,0}, k_{v,1} = k_{v,0} \oplus \Delta^v$ and permutation bits $\lambda_u, \lambda_v \in \{0,1\}$. The parties perform the procedure $\mathsf{Solder(u,v)}$ defined as follows: If $\Lambda_u = 0$ the parties collaboratively compute

$$s^0_{u\to v} = \Big((\lambda_u \oplus (1-\lambda_v)) \cdot (k_{u,0} \oplus k_{v,0})\Big) \oplus \Big((\lambda_u \oplus \lambda_v) \cdot (k_{u,0} \oplus k_{v,1})\Big)$$

and output $s^0_{u\to v}$ to everyone.

Otherwise, if $\Lambda_u = 1$ the parties collaboratively compute

$$s^1_{u\to v} = \Big((\lambda_u \oplus (1-\lambda_v)) \cdot (k_{u,1} \oplus k_{v,1})\Big) \oplus \Big((\lambda_u \oplus \lambda_v) \cdot (k_{u,1} \oplus k_{v,0})\Big)$$

and output $s^1_{u\to v}$ to everyone.

The information $s^{\Lambda_u}_{u\to v}$ allows the parties to solder wire $v$ to wire $u$. Notice that this technique involves only one multiplication layer, since the parties simultaneously compute both multiplications and then *locally* add results in $\mathbb{F}_{2^\kappa}$.

Observe that our variation of the soldering is applicable to the multiparty settings as well, in addition, due to the already exist SPDZ shares to the full wires' keys, we don't need to rely on additional homomorphic commitment scheme and its expensive overhead. Moreover, the original soldering was thought to be a way to connect two wires *within the same circuit* (using a single global difference) while in here we show that it is applicable for wires *of different circuits* as well (that were garbled independently and with two global differences $\Delta^u, \Delta^v$).

To see why it works, without loss of generality, consider $\Lambda_u = 0$. If $\lambda_u = \lambda_v$, then given key $k_{u,0}$ that carries value $\rho_u = \lambda_u$, the parties compute

$$\begin{aligned} k_{u,0} \oplus s^0_{u\to v} &= k_{u,0} \oplus \big(1 \cdot (k_{u,0} \oplus k_{v,1})\big) \oplus \big(0 \cdot (k_{u,0} \oplus k_{v,0})\big) \\ &= k_{v,0} \end{aligned}$$

such that $k_{u,0}$ and $k_{v,0}$ encapsulate the same real value as required. If $\lambda_u \neq \lambda_v$ we get

$$\begin{aligned} k_{u,0} \oplus s^0_{u\to v} &= k_{u,0} \oplus \big(0 \cdot (k_{u,0} \oplus k_{v,0})\big) \oplus \big(1 \cdot (k_{u,0} \oplus k_{v,1})\big) \\ &= k_{v,1} \end{aligned}$$

such that $k_{u,0}$ and $k_{v,1}$ encapsulate the same real value as required. The same analysis holds when $\Lambda_u = 1$.

*Performance.* To obtain the key of the next circuit the parties simultaneously compute 2 multiplications (of $n$ keys) over the shares in one round and then open the result in the second round, hence $2n$ multiplication triples are required in the offline phase. Multiplication requires the communication of $3n\kappa$ bits per party, opening requires $\kappa$, a total of $(3n + 1)\kappa$ per party per input bit.

### 4.3 Memory via Free Conversion Between Keys and Shared Real Values

In this section we present a new technique, which outperforms both the embedding and soldering techniques in both the communication and memory size overheads. Essentially, it allows to freely convert between BMR wire keys and SPDZ secret shares of the real values that those keys represent. As before, it is not necessary to know which SPDZ share to convert from and to in garbling time. This allows reactive memory accesses in the sense that, during evaluation, the parties can evaluate previously garbled circuits on values read from memory at an address that was only just revealed during the evaluation phase. The latter is crucial for implementing ORAM.

Using this technique the parties need to compute only (local) additions and some SPDZ openings in order to move from the evaluation of one circuit to the next. In more detail, converting from wire keys to SPDZ shares can be done without communication at all, while the other direction requires two rounds of SPDZ opening. In any case, no multiplication is necessary, hence, no offline overhead (for triple generation) is implied. Similarly, the information required is a by-product of the BMR offline phase, hence there is no extra cost there.

When reading a bit from memory the parties need to know the external value of the wire associated with it. For a circuit-input-wire, which is associated with a particular party, that party knows the wire's permutation and external value, hence, it can broadcast the external value to the parties, who then can broadcast their appropriate share of the BMR key. In contrast, when reading from memory, the external value of the wire is shared (it is nobody's input) and reconstructed, then the parties broadcast their keys as before. The keys that the parties broadcast are stored by each party along with the garbled circuit, in the same streaming manner, and are not part of the program's memory.

*Packing secret bits.* Naively storing each external value as a SPDZ secret share would require $s^2$ bits in memory for every $s$-bit data block. We can reduce this overhead by packing $s$ secret shares of bits into an $s$-bit secret share such that it requires only $2s$ bits in memory for every $s$-bit data block ($s$ bits for the share itself and another $s$ bits for its MAC). Packing $s$ bits $[\![b_0]\!], \ldots, [\![b_{s-1}]\!]$ is done by computing $[\![B]\!] = \sum_{i\in[s]} b_i \cdot 2^i$ where the $2^i$ part is constant, so we obtain $[\![B]\!]$ by local computation only. Now, we can make operations over bits easily by inputting the entire data item and using the specific required bit. "Extracting" the $j$-th bit, $[\![b_j]\!]$, from $[\![B]\!]$ can be done locally as well as described below.

**Writing to Memory.** Recall that after issuing the **Garble** instruction in $\mathcal{F}_{\mathsf{BMR}}$ the parties hold shares to all the permutation bits of all wires. Recall that the wires groups $W^t_{\mathsf{addr,wr}}$ and $W^t_{\mathsf{write}}$ refer to the wires associated with the address to be written and the value to be written to that address respectively. In the protocol, the parties open the permutation bits for wires $W^t_{\mathsf{addr,wr}}$ but not for wires $W^t_{\mathsf{write}}$, this means that they learn $i^{\mathsf{write}}_t$ in the clear, but learn nothing about $b^{\mathsf{write}}_t$, rather, they only obtain the keys and their external values associated with it. That is, for $w \in W^t_{\mathsf{write}}$, the parties obtain $k_{w,\Lambda_w}$ and $\Lambda_w$. To store the *real* value that is carried by wire $w$ in memory address $i$ the parties only need to compute $[\![\rho_w]\!] = \Lambda_w + [\![\lambda_w]\!]$. Then $p_i$ stores $D[i] \leftarrow [\![\rho_w]\!]^i$. Furthermore, every party $p_i$ can check whether $k_{w,\Lambda_w}$ is correct because they have obtained both $k_{w,0}$ and $k_{w,1} = k_{w,0} \oplus \Delta_i$ during **Garble**. This is equivalent to checking the correctness of output wires. In order to achieve optimal memory usage, $s$ bits $w_0, \ldots, w_{s-1}$ can be combined by (locally) computing $\sum_{i=0}^{s-1} X^i [\![\rho_{w_i}]\!]$ where $X$ denotes a generator of the multiplicative group of a field of size $2^s$.

**Reading from Memory.** Let $i^{\mathsf{read}}_{t-1}$ be the address from which the parties are instructed to read when evaluating $GC^{t-1}$ and let $w \in W^t_{\mathsf{read}}$. We assume for a moment that secret shares packing technique above has not been applied when storing. Therefore, $D[i^{\mathsf{read}}_{t-1}]$ contains a share of the bit $[\![\rho_{w'}]\!]$ that was most recently written to $i^{\mathsf{read}}_{t-1}$ at a previous timestep $t'$ with wire $w' \in W^{t'}_{\mathsf{write}}$. Party $p_i$ holds both $k^i_{w,0}$ and $k^i_{w,1}$, but need to broadcast only one of them. Specifically, broadcast $k^i_{w,\Lambda_w}$ for $\Lambda_w = \rho_w \oplus \lambda_w$. Now, since we require that $\rho_w = \rho_{w'}$ then the parties open $\Lambda_w = [\![\rho_w]\!] + [\![\lambda_w]\!] = [\![\rho_{w'}]\!] + [\![\lambda_w]\!]$ and broadcast $k_{w,\Lambda_w}$. Finally, if the parties have stored $\sum_{i=0}^{s-1} X^i [\![\rho_{w_i}]\!]$ at a particular memory address, $[\![\rho_{w_i}]\!]$ can be computed by opening $\sum_{i=0}^{s-1} X^i [\![\rho_{w_i}]\!] + \sum_{i=0}^{s-1} X^i [\![\lambda_{w_i}]\!]$. This works because any field of size $2^s$ has characteristic two, thus addition corresponds to bitwise XOR.

# 5 Realizing Functionality $\mathcal{F}_{\mathsf{UAP}}$

Protocol $\mathcal{P}_{\mathsf{UAP}}$ in Fig. 7 realizes $\mathcal{F}_{\mathsf{UAP}}$ in the $\mathcal{F}_{\mathsf{BMR}}$-hybrid model.

**Protocol** $\mathcal{P}_{\mathsf{UAP}}$

Party $p_i$ allocates a memory $D^i$ of size $N$.

**Offline.** The parties initialize $\mathcal{F}_{\mathsf{MPC}}$ by calling $\mathcal{F}_{\mathsf{MPC}}.\mathsf{Init}(k)$. The parties run $\mathcal{F}_{\mathsf{BMR}}.\mathsf{Garble}(\mathrm{C}_{\mathrm{CPU}}, (W, P))$ to garble $T$ CPU circuits: $GC^1, \ldots, GC^T$, where $(W, P)$ maps from input wires to their associated party in the first execution and are the empty set in all other invocations.

**Input.** To input $x^i = x^i_1, \ldots, x^i_\ell$ of party $p_i$, let $w$ be the wire associated with input bit $x^i_j$. Party $p_i$ broadcasts $\Lambda_w = x^i_j \oplus \lambda_w$ and every party $p_j$ broadcasts $k^j_{w,\Lambda_w}$.

**Output.** Invoke $\mathcal{F}_{\mathsf{BMR}}.\mathsf{Open}(GC^t)$ for all $t \in [T]$.
For $t = 1$ to $T$:

1. The parties run $\mathcal{E}_{\mathsf{BMR}}(GC^t)$ and obtain $k_{w,\Lambda_w}, \Lambda_w$ for every $w \in W^t_{\mathsf{write}}$ and obtain $i^{\mathsf{write}}_t, i^{\mathsf{read}}_t$ in the clear.
2. For each $w \in W^t_{\mathsf{write}}$: Locally compute $[\![\rho_w]\!] = \Lambda_w + [\![\lambda_w]\!]$. Party $p_i$ stores $[\![\rho_w]\!]^i$ in $D^i[i^{\mathsf{write}}_t]$. In addition, they check whether $k_{w,\Lambda_w}$ equals the key received in $\mathcal{P}_{\mathsf{BMR}}.\mathsf{Garble}$.
3. For each $w \in W^{t+1}_{\mathsf{read}}$: Let $i^{\mathsf{read}}_t$ be the memory address from which $w$ is to be fed. Then the parties open $\Lambda_w = D[i^{\mathsf{read}}_t] + [\![\lambda_w]\!]$. Given $\Lambda_w$, party $p_i$ broadcasts $k^i_{w,\Lambda_w}$ so that all parties finally hold $k_{w,\Lambda_w}$.
4. Obtain $k_{w,\Lambda_w}$ for every $w \in W^{t+1}_{\mathsf{state}}$.

Finally, for every $w \in W^T_{\mathsf{state}'}$ (output wires of $\mathrm{C}^T_{\mathrm{CPU}}$) the parties open $\lambda_w$ and compute $\rho = \lambda_w \oplus \Lambda_w$.

**Fig. 7.** Realizing $\mathcal{F}_{\mathsf{UAP}}$ in the $\mathcal{F}_{\mathsf{BMR}}$-hybrid model.

## 5.1 Security of Protocol $\mathcal{P}_{\mathsf{UAP}}$

The security of our construction relies on the security of the underlying BMR and SPDZ protocols and the security of the transformation between the garbled wires and SPDZ shares. Informally, the latter can be seen as follows: Neither transformation reveals any secret information because one direction (writing to memory) is done locally, and the other one (reading form memory) only reveals an external value and the corresponding wire label, both of which hide the real value that is carried over the wire according to the security of the BMR protocol. For malicious security, consider that revealing the external value is done using SPDZ, which guarantees correctness by checking the MAC. Furthermore, if any party broadcasts a faulty share of the BMR key, this is guaranteed to lead to an invalid output key (and thus easy detection by honest parties) by the properties of the BMR protocol. More formally, we prove the following theorem:

**Theorem 5.1.** *Protocol* $\mathcal{P}_{\mathsf{UAP}}$ *(Fig. 7) realizes functionality* $\mathcal{F}_{\mathsf{UAP}}$ *(Fig. 2) in the* $\mathcal{F}_{\mathsf{BMR}}$*-hybrid model.*

*Proof.* Let $\mathcal{A}$ be an adversary controlling a subset of the parties, denoted $A = \{p_{i_1}, \ldots, p_{i_c}\}$ and denote by $\bar{A} = [n] \setminus A$ the subset of the honest parties.

We present a simulator $\mathcal{S}$ who participates in the ideal execution $\mathcal{F}_{\mathsf{UAP}}$ by taking the role of $A$ and in an internal execution of $\mathcal{P}_{\mathsf{UAP}}$ with $\mathcal{A}$, in which $\mathcal{S}$ takes the role of $\bar{A}$ and the functionality $\mathcal{F}_{\mathsf{BMR}}$. The simulator $\mathcal{S}$ uses another simulator $\mathcal{S}_{\mathsf{BMR}}$ that when given the adversary's input/output to/from a circuit, and both keys to all wires in the garbled circuit, produces a view that is indistinguishable to the view of the adversary's evaluation of the circuit in the real execution (such a simulator was presented in [25]).

The simulator $\mathcal{S}$ does as follows:

1. **Extract $\mathcal{A}$'s inputs**
   (a) In the internal execution, garble $T$ copies of $\mathrm{C}_{\mathrm{CPU}}$ exactly as described in $\mathcal{F}_{\mathsf{BMR}}$. In particular, for every input wire $w \in GC^1$ associated with a corrupted party $p_c \in A$ output $\lambda_w$ in the clear to $p_c$.
   (b) Upon issuing the **Input** command in the internal execution, for an input wire $w$ associated with a corrupted party $p_c \in A$, receive $p_c$'s external value $\Lambda_w$ and compute $p_c$'s input to wire $w$ by $\rho_w = \Lambda_w \oplus \lambda_w$ (the simulator $\mathcal{S}$ knows $\lambda_w$ because it was garbling the circuit on behalf of $\mathcal{F}_{\mathsf{BMR}}$).
2. Engage in the ideal execution $\mathcal{F}_{\mathsf{UAP}}$ by inputting the values extracted above as the corrupted parties' input and obtain $\boldsymbol{y} = \Pi(D, \boldsymbol{x})$ along with $\mathsf{AP}(\Pi, D, \boldsymbol{x})$.
3. Open all garbled circuits $GC^1, \ldots, GC^T$ toward the adversary.
4. **Evaluation**
   (a) Invoke $\mathcal{S}_{\mathsf{BMR}}$ with $\mathcal{A}$'s inputs that were extracted earlier, the garbled circuit $GC^1$ and the adversary's output from $GC^1$: $\mathsf{access}_1 = (i_1^{\mathsf{read}}, i_1^{\mathsf{write}})$.[6] Output whatever $\mathcal{S}_{\mathsf{BMR}}$ produced.
   (b) Note that $A$ have no inputs to circuits $GC^2, \ldots, GC^T$, thus, for every such circuit we invoke $\mathcal{S}_{\mathsf{BMR}}$ with no inputs at all. Then, for every input wire of the garbled circuit, $\mathcal{S}$ checks which of its keys $k_{w,0}$ or $k_{w,1}$ was produced as the simulated view, and then supply the correct share of the external value of that key ($\mathcal{S}$ can do this since it knows the global MAC key used for the SPDZ shares). Formally: For $t = 2$ to $T$:
      i. Output $\mathsf{access}_{t-1} = i_{t-1}^{\mathsf{read}}, i_{t-1}^{\mathsf{write}}$ in the clear.
      ii. Invoke $\mathcal{S}_{\mathsf{BMR}}$ with $GC^t$. If $t = T$ then supply $\mathcal{S}_{\mathsf{BMR}}$ with $\boldsymbol{y}$ as well.
      iii. For every input wire $w$ of $GC^t$, extract from the produced view the key $k_{w,\Lambda_w}$ used in the evaluation.

[6] Note that $\mathsf{access}_1$ is the output to all parties, not only the adversary's, however, for the simulation purpose we use this as the adversary's output.

- iv. For every input wire $w$ of $GC^t$ simulate the opening of $[\![\Lambda_w]\!] = [\![\lambda_w \oplus \rho_w]\!]$ (from Step 3 in $\mathcal{P}_{\mathsf{UAP}}$) as follows: Let the shares of $A$ be $\Lambda_w^A$, then $\mathcal{S}$ chooses random shares $\Lambda_w^h$ for every honest party $p_h \in \bar{A}$ such that $\Lambda_w = \Lambda_w^A + \sum_{h \in \bar{A}} \Lambda_w^h$ and use them to open $\Lambda_w$.
- v. Output the view produced by $\mathcal{S}_{\mathsf{BMR}}$ in Step 4(b)ii above.

*Claim.* For every PPT $\mathcal{A}$ and for every $\boldsymbol{x}$ the output of $\mathcal{S}$ above is indistinguishable to the view of $\mathcal{A}$ in the real execution of $\mathcal{P}_{\mathsf{UAP}}$.

*Proof.* We define hybrid $\mathbf{Hyb}_t$ to be as follows: The adversary view in the real execution of $\mathcal{P}_{\mathsf{UAP}}$ for timesteps $1, \ldots, t$, followed by the simulated view for timesteps $t+1, \ldots, T$ as described in Step 4b of the simulation above. Thus, $\mathbf{Hyb}_T$ is exactly the real execution of $\mathcal{P}_{\mathsf{UAP}}$ and $\mathbf{Hyb}_0$ is exactly the output of $\mathcal{S}$ described above. Assume by contradiction that there exists a PPT $\mathcal{D}$ who can distinguish between $\mathbf{Hyb}_T$ and $\mathbf{Hyb}_0$ with non negligible probability, then we construct $\mathcal{D}'$ who distinguishes between a real execution of $\mathcal{P}_{\mathsf{BMR}}$ to the output of $\mathcal{S}_{\mathsf{BMR}}$ (for a single circuit) as follows: By the existence of $\mathcal{D}$ it is implied that there exists $t'$ for which $\mathcal{D}$ distinguishes between $\mathbf{Hyb}_t$ and $\mathbf{Hyb}_{t+1}$ with non negligible probability. Then, given a circuit $C$ and a view $V$ which is either the view of the adversary in a real execution of $\mathcal{P}_{\mathsf{BMR}}$ or the output of $\mathcal{S}_{\mathsf{BMR}}$, $\mathcal{D}'$ generates a real view of the execution of $\mathcal{P}_{\mathsf{UAP}}$ for timesteps $1, \ldots, t$, then plugs $V$ together with the opening of external values of the input wires of $C$, and then complete the simulation according to Step 4b above. Finally, hands the result view to $\mathcal{D}$ and outputs whatever $\mathcal{D}$ outputs. Observe that if $V$ is a view of a real execution then the above is distributed exactly as $\mathbf{Hyb}_t$, otherwise, it is distributed exactly as $\mathbf{Hyb}_{t+1}$. It follows that $\mathcal{D}'$ distinguishes between the real execution of $\mathcal{P}_{\mathsf{BMR}}$ and the output of $\mathcal{S}_{\mathsf{BMR}}$ with non negligible probability, by contradiction of the security of $\mathcal{P}_{\mathsf{BMR}}$.

## 6 Optimizing BMR Evaluation

The free-XOR technique of [2] makes space and communication complexities linear in the number of AND gates (XOR gates are almost for free[7]). In this section we show how to further decrease memory consumption in the online phase by a factor of up to 2. Even though our technique could be applied to a plain BMR protocol, we present the idea over a scheme that uses the free-XOR. We stress that it is not limited to secure *RAM* computation but also applicable in BMR-based protocols, even with only a single execution.

The $\mathsf{Evaluate}$ instruction in $\mathcal{F}_{\mathsf{BMR}}$ (Fig. 3) that is invoked in the online phase traverses the circuit in a topological order and obtains a single output key $k_w \in (\mathbb{F}_{2^\kappa})^n$ for every wire $w$ in the circuit, until it reaches the output wires. To check the authenticity of $k_w$, party $p_i$ extracts the $i$th element, $k_w^i \in \mathbb{F}_{2^\kappa}$, and verifies that it is one of the keys given to him by $\mathcal{F}_{\mathsf{MPC}}$ in the offline phase, that is, $k_w^i \in \{k_{w,0}^i, k_{w_1}^i\}$. In case that $k_w^i \notin \{k_w^0, k_w^1\}$ then $p_i$ notifies all parties with

[7] They require only a simple XOR operation.

regard to the corrupted garbled circuit and aborts. Using our technique, it is possible for $p_i$ to discard its keys $\{k^i_{w,0}, k^i_{w_1}\}$ of all wires right after the garbled circuit construction is complete (in the offline phase), instead, it has to store only a *single* bit per wire. Since the garbled gate is of size $4nk$ and the original verification procedure requires memory of size $2k$ (i.e. party $p_i$ stores the two keys of the output wire of the gate), this results with a decrease of memory consumption by a factor of $\frac{1}{2n}$. However, a great improvement is achieved for a more recent construction [3]. In that construction the size of a garbled gate is $4k$ (i.e. it is independent of the number of parties $n$), thus, memory saving is significant.

Using our technique, the evaluator is saved from loading and comparing 1.5 keys per wire in average (since in half of the wires the verification passes after the first comparison). This loading[8] and comparison time became substantial as the computation of AES has been considerably improved[9].

### 6.1 The Technique

Circuit garbling is done in the offline phase of the protocol using the $\mathcal{F}_{\mathsf{MPC}}$ functionality (Fig. 11). Let $\mathsf{lsb}(x)$ denote the least significant bit of $x$. We instruct $\mathcal{F}_{\mathsf{MPC}}$ to choose $\varDelta = (\varDelta_1, \ldots, \varDelta_n)$ such that $\mathsf{lsb}(\varDelta_i) = 1$ for every $i \in [n]$. The result is that $\mathsf{lsb}(k^i_{w,0}) \neq \mathsf{lsb}(k^i_{w,1})$ for all $w$ and $i$. When garbling is completed using $\varDelta$ as described, party $p_i$ stores the bit $\delta^i_w = \mathsf{lsb}(k^i_{w,0})$ for every wire $w$. In addition party $p_i$ discards all keys $k^i_{w,0}$ and $k^i_{w,1}$ for all but the output wires.

The evaluation of the circuit is done exactly as before, however, instead of verifying the key validity of the output wire of *every* gate, this is done only for *output* gates. For an inner gate with output wire $w$, party $p_i$ obtains the external value $\varLambda_w$ by computing $\varLambda_w = \mathsf{lsb}(k^i_w) \oplus \delta^i_w$. This way the parties learn that the key $k^i_w$ obtained by evaluating a gate is actually the $\varLambda$-key. For output gates (i.e. gates whose output wire is also a circuit-output wire), party $p_i$ verifies that $k^i_w \in \{k^i_{w,0}, k^i_{w,1}\}$ as before.

Forcing the last bit of a random element is featured in SPDZ-like implementation of $\mathcal{F}_{\mathsf{MPC}}$ (e.g. [23]) since they are inherently bit wise, so we can generate $k-1$ random bits and then compose the field element accordingly so its last bit is 1.

### 6.2 Security

Notice that we use the exact same garbling procedure as in [2] except that here the last bit of every $\varDelta_i$ is known to the adversary (i.e. $\mathsf{lsb}(\varDelta_i) = 1$) whereas in their scheme *all* bits of $\varDelta$ are random. The security of our scheme can be easily reduced to the security of [2]. Our simulator is the same simulator as in [2]. Let the distinguisher's advantage in distinguishing between the real execution

[8] Loading time depends on the implementation, i.e. whether using dereferences or not.

[9] Using the AES-NI instruction set from Intel's Sandy Bridge microarchitecture and on, a RoundKey instruction takes a single CPU cycle and latency of 8, that is, one could reach a throughput of up to 8 RoundKey operations with the same key at the same CPU cycle [21, Chap. 5.10].

of our scheme to the ideal execution be $\epsilon$. Then, the advantage of the *same* distinguisher in distinguishing between the real execution and the simulation of [2] is $\epsilon' = \epsilon \cdot \frac{1}{2^h}$ for $h$ honest parties. This holds because the probability of having $\mathsf{lsb}(\varDelta_i) = 1$ in the free-XOR scheme is $\frac{1}{2}$ for an honest party $p_i$. Recall that in the original scheme, the security depends on $h$ keys of length $k$. Thus, increasing the advantage of the adversary by $2^h$ is negligible. Assuming that [2] is secure we conclude that our scheme is secure as well.

## 7 Implementation

In this section we report our results of the *first* (to the best of our knowledge) implementation of a garbled-circuit-based secure RAM computation for setting with active security and dishonest majority. We chose to implement our third technique (Sect. 4.3) as it is the most efficient technique for memory access. We have combined our new BMR implementation with the existing SPDZ system [6], and used it to implement an oblivious array[10] using Circuit ORAM [36]. The code is written in C++ using the AES-NI and AVX2 instruction sets.

**Experiments.** Our timing results below refer to the following experiments:

1. Circuit ORAM [36] using the BMR-SPDZ protocol with the scheme in Sect. 4.3, labeled as 'BMR, Circuit ORAM' in the figures below.
2. Circuit ORAM [36] using a pure SPDZ implementation, labeled as 'Pure SPDZ, Circuit ORAM'.
3. Path ORAM [35] using a pure SPDZ implementation [24], labeled as 'Pure SPDZ, Path ORAM'.
4. Trivial ORAM, i.e. linear scanning of the entire memory for every access, labeled as 'BMR, linear scan'.

The Path ORAM intends to optimize the *bandwidth cost* and *bandwidth blowup* where bandwidth cost refers to the average number of bits transferred for accessing a single block and bandwidth blowup is defined as bandwidth cost divided by the block size (i.e., the bit-length of a data block)[11]. The results by Keller and Scholl [24] are reported using Path ORAM, which seems preferable when round complexity is not a concern. For the sake of comparison, we have also implemented Circuit ORAM using pure SPDZ. Comparing experiments (1), (2) and (3) in Figs. 8 and 9, our approach outperforms the pure SPDZ when the parties are connected over a WAN, independently of the choice of the ORAM scheme. Furthermore, experiment (4) allows to find the *breakeven* points, that is,

[10] "Oblivious array" is the name given in [24] to the basic oblivious random memory access, which allows reading and writing with a secret index. This is in distinction to "oblivious dictionary" that allows reading according to a secret 'key' in a key-value (dictionary) data structure, where the key may be larger than the size of the memory.

[11] As defined in [36, A.2] under ORAM metrics.

to figure out up to what memory size the linear scan performs better than applying an ORAM algorithm. Given the simplicity of a linear scan, it is clear that it is faster for small enough sizes.

All experiments were performed for both LAN and WAN environment to test the influence of our approach of reducing the round complexity. We stress that our implementation is the first in this setting even when considering 2 parties only. Nevertheless, we report timing results for a protocol with 3 participants as well.

**Parameters.** Our security parameters are $\kappa = 128$ and $s = 40$. In all experiments, the oblivious arrays are made up of 32-bit entries, and all figures refer to the array size as the number of such entries. Therefore, our figures range from $1024 \cdot 32 \approx 32\,\text{kB}$ to $2^{25} \cdot 32 \approx 1.1\,\text{GB}$.

Our ORAM implementations (Circuit ORAM and Path ORAM) require up to three recursions such that intermediate ORAMs use 128-bit entries, and we use a linear scan for less than 256 such entries.

All reported results are measured per *logical access* to the memory (array), which, as explained before, may incorporate many physical accesses.

**Environment.** Our implementations were done using 4th generation Intel Core i7 with 8 cores running at 3.5 GHz, 16 GB RAM, and SSD (to store the garbled circuits) connected over a LAN (bandwidth of 1 Gbit/s and RTT of 0.1–0.2 ms).

Furthermore, we have simulated a WAN setting on the same machines by extending the round trip time to 100 ms and restricting the throughput to 50 Mbit/s. Figure 8 shows our results for the two settings with two parties while Fig. 9 shows our results for three parties. They confirm that using garbled circuits (BMR) is beneficial with high network latencies. With BMR, combining Circuit ORAM with our memory access surpasses linear scanning below a size of one million.

**Offline Cost.** Finally, for a more complete picture, we have estimated the offline cost in the LAN setting. Figure 10 shows the cost for one access of Circuit ORAM

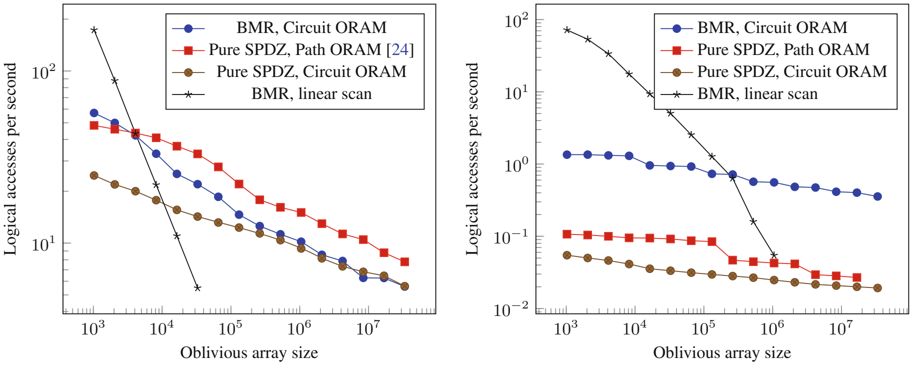

**Fig. 8.** Two parties over LAN (left) and over WAN (right).

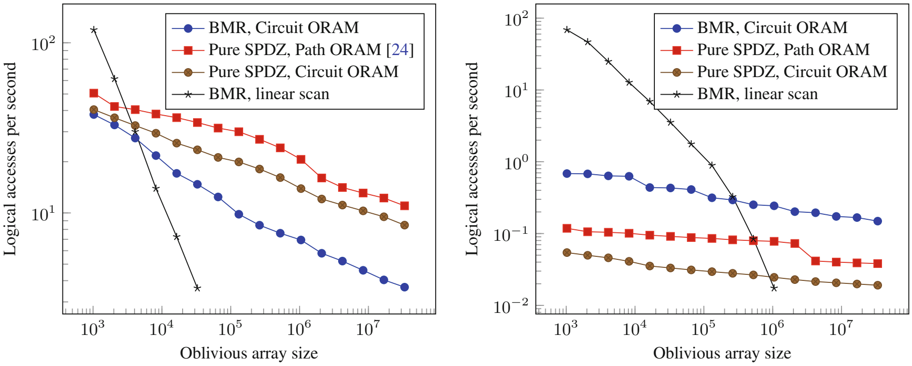

**Fig. 9.** Three parties over LAN (left) and over WAN (right).

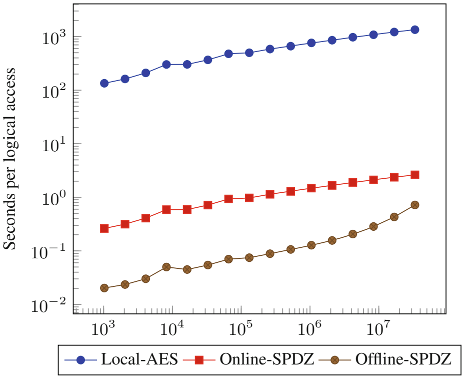

**Fig. 10.** Offline time per logical access with Circuit ORAM and BMR.

implemented in BMR. All figures are based on the number of AND gates in the circuit computing Circuit ORAM because the preprocessing information required for soldering is essentially a by-product of the circuit generation.

To get a better picture of the offline performance of our protocol, we separated it into three parts:

- **Offline-SPDZ.** This is the offline phase of the SPDZ protocol, which is independent of the circuit the parties wish to evaluate. In this phase the parties produce the multiplication triples that would be required for the garbling. The numbers in this part are based on a production of 4828 triples per second as reported by Keller et al. [23].
- **Local-AES.** Local computation of AES ciphers. The parties use the results of that computation as input to the Online-SPDZ part, which use them in order to construct the garbled circuit.

- **Online-SPDZ.** This is the online phase of the SPDZ protocol, in which the parties evaluate a circuit that garbles the actual circuit they want to evaluate in the BMR online phase.

In the figure we can easily observe that Offline-SPDZ dominates the cost by 3–4 orders of magnitudes because of the communication cost of MASCOT [23].

## A The Generic Reactive MPC Functionality

The following functionality is used by protocols that follow the BMR-SPDZ approach.

**The Generic Reactive MPC Functionality:** $\mathcal{F}_{\mathsf{MPC}}$

The functionality consists of seven externally exposed commands **Initialize**, **InputData**, **RandomBit**, **Random**, **Add**, **Multiply**, and **Output**, and one internal subroutine **Wait**.

**Initialize** On input $(\mathsf{Init}, k)$ from all parties, the functionality activates and stores $k$. All additions and multiplications below will be in $\mathbb{F}_{2^\kappa}$.

**Wait** This waits on the adversary to return a $GO/NO\text{-}GO$ decision. If the adversary returns $NO\text{-}GO$ then the functionality aborts.

**InputData** On input $(\mathsf{Input}, p_i, varid, x)$ from party $p_i$ and $(input, p_i, varid, ?)$ from all other parties, with $varid$ a fresh identifier, the functionality stores $(varid, x)$. The functionality then calls **Wait**.

**RandomBit** On command $(\mathsf{RandomBit}, varid)$ from all parties, with $varid$ a fresh identifier, the functionality selects a random value $r \in \{0, 1\}$ and stores $(varid, r)$. The functionality then calls **Wait**.

**Random** On command $(\mathsf{Random}, varid)$ from all parties, with $varid$ a fresh identifier, the functionality selects a random value $r \in \mathbb{F}_{2^\kappa}$ and stores $(varid, r)$. The functionality then calls **Wait**.

**Add** On command $(\mathsf{Add}, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, x + y)$. The functionality then calls **Wait**.

**Multiply** On input $(\mathsf{Multiply}, varid_1, varid_2, varid_3)$ from all parties (if $varid_1, varid_2$ are present in memory and $varid_3$ is not), the functionality retrieves $(varid_1, x)$, $(varid_2, y)$ and stores $(varid_3, xy)$. The functionality then calls **Wait**.

**Output** On input $(\mathsf{Output}, varid, i)$ from all honest parties (if $varid$ is present in memory), the functionality retrieves $(varid, x)$ and outputs either $(varid, x)$ in the case of $i \neq 0$ or $(varid)$ if $i = 0$ to the adversary. The functionality then calls **Wait**, and only if **Wait** does not abort then it outputs $x$ to all parties if $i = 0$, or it outputs $x$ only to party $i$ if $i \neq 0$.

**Fig. 11.** The generic reactive MPC functionality

## References

1. Afshar, A., Hu, Z., Mohassel, P., Rosulek, M.: How to efficiently evaluate RAM programs with malicious security. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 702–729. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_27
2. Ben-Efraim, A., Lindell, Y., Omri, E.: Optimizing semi-honest secure multiparty computation for the internet, pp. 578–590 (2016)
3. Ben-Efraim, A., Lindell, Y., Omri, E.: Efficient scalable constant-round MPC via garbled circuits. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017. LNCS, vol. 10625, pp. 471–498. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70697-9_17
4. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 337–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_12
5. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing: improvements and extensions, pp. 1292–1303 (2016)
6. Bristol Cryptography Group: SPDZ software (2016). https://www.cs.bris.ac.uk/Research/CryptographySecurity/SPDZ/
7. Canetti, R., Holmgren, J.: Fully succinct garbled RAM, pp. 169–178 (2016)
8. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_38
9. Doerner, J., Shelat, A.: Scaling ORAM for secure computation. In: CCS (2017)
10. Frederiksen, T.K., Jakobsen, T.P., Nielsen, J.B., Nordholt, P.S., Orlandi, C.: MiniLEGO: efficient secure two-party computation from general assumptions. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 537–556. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_32
11. Garg, S., Gupta, D., Miao, P., Pandey, O.: Secure multiparty RAM computation in constant rounds. In: Hirt, M., Smith, A. (eds.) TCC 2016. LNCS, vol. 9985, pp. 491–520. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53641-4_19
12. Garg, S., Lu, S., Ostrovsky, R.: Black-box garbled RAM, pp. 210–229 (2015)
13. Garg, S., Lu, S., Ostrovsky, R., Scafuro, A.: Garbled RAM from one-way functions, pp. 449–458 (2015)
14. Gentry, C., Halevi, S., Lu, S., Ostrovsky, R., Raykova, M., Wichs, D.: Garbled RAM revisited. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 405–422. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_23
15. Goldreich, O.: Foundations of Cryptography: Basic Applications, vol. 2. Cambridge University Press, New York (2004)
16. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC, pp. 218–229 (1987)
17. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**(3), 431–473 (1996)
18. Gordon, S.D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., Vahlis, Y.: Secure two-party computation in sublinear (amortized) time, pp. 513–524 (2012)

19. Hazay, C., Yanai, A.: Constant-round maliciously secure two-party computation in the RAM model. In: Hirt, M., Smith, A. (eds.) TCC 2016. LNCS, vol. 9985, pp. 521–553. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53641-4_20
20. Hu, Z., Mohassel, P., Rosulek, M.: Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 150–169. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48000-7_8
21. Intel: Intel 64 and IA-32 Architectures Optimization Reference Manual (2016). http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html
22. Keller, M.: The oblivious machine - or: how to put the C into MPC. Cryptology ePrint Archive, Report 2015/467 (2015). http://eprint.iacr.org/2015/467
23. Keller, M., Orsini, E., Scholl, P.: MASCOT: faster malicious arithmetic secure computation with oblivious transfer, pp. 830–842 (2016)
24. Keller, M., Scholl, P.: Efficient, oblivious data structures for MPC. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8874, pp. 506–525. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45608-8_27
25. Lindell, Y., Pinkas, B., Smart, N.P., Yanai, A.: Efficient constant round multi-party computation combining BMR and SPDZ. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 319–338. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48000-7_16
26. Lindell, Y., Riva, B.: Blazing fast 2PC in the offline/online setting with security for malicious adversaries. In: CCS, pp. 579–590 (2015)
27. Lindell, Y., Smart, N.P., Soria-Vazquez, E.: More efficient constant-round multi-party computation from BMR and SHE. In: Hirt, M., Smith, A. (eds.) TCC 2016. LNCS, vol. 9985, pp. 554–581. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53641-4_21
28. Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.W.: Automating efficient RAM-model secure computation, pp. 623–638 (2014)
29. Liu, C., Wang, X.S., Nayak, K., Huang, Y., Shi, E.: ObliVM: a programming framework for secure computation, pp. 359–376 (2015)
30. Lu, S., Ostrovsky, R.: How to garble RAM programs? In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 719–734. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_42
31. Mohassel, P., Rosulek, M., Scafuro, A.: Sublinear zero-knowledge arguments for RAM programs. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10210, pp. 501–531. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56620-7_18
32. Nielsen, J.B., Orlandi, C.: LEGO for two-party secure computation. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 368–386. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00457-5_22
33. Ostrovsky, R., Shoup, V.: Private information storage (extended abstract), pp. 294–303 (1997)
34. Shi, E., Chan, T.-H.H., Stefanov, E., Li, M.: Oblivious RAM with $O((\log N)^3)$ worst-case cost. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 197–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25385-0_11
35. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C.W., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol, pp. 299–310 (2013)

36. Wang, X., Chan, T.-H.H., Shi, E.: Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound, pp. 850–861 (2015)
37. Wang, X., Malozemoff, A.J., Katz, J.: Faster secure two-party computation in the single-execution setting. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10212, pp. 399–424. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56617-7_14
38. Wang, X., Ranellucci, S., Katz, J.: Authenticated garbling and efficient maliciously secure two-party computation. In: CCS, pp. 21–37 (2017)
39. Wang, X.S., Gordon, S.D., McIntosh, A., Katz, J.: Secure computation of MIPS machine code. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9879, pp. 99–117. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45741-3_6
40. Yao, A.C.-C.: How to generate and exchange secrets (extended abstract). In: FOCS, pp. 162–167 (1986)