AN ABSTRACT OF THE THESIS OF

Zhangxiang Hu for the degree of Master of Science in Computer Science presented on June 3, 2015.

Title: RANDOM ACCESS MACHINE IN SECURE MULTI-PARTY COMPUTATION

Abstract approved: _____

Michael J. Rosulek

Secure multi-party computation (MPC) is a conceptual framework in cryptography. It allows distrusting parties engage in a protocol to perform a computational task while still maintain some secure properties. Most existing approaches are required to interpret functions as a boolean circuit. With the recent state-of-art circuit garbling scheme, the performance are significantly improved. However, boolean circuit still has its limitations in practical usage, especially when the input data size is enormous.

In this thesis, we focus on another technique in MPC which is called *random-access machines* (RAM program). We first describe a zero-knowledge proof system in which a prover holds a large dataset M and can repeatedly prove NP relations about that dataset. This system achieves sublinear amortized cost. Second, we present the first practical protocols for evaluating RAM programs with security against malicious adversaries. The extra overhead of obtaining malicious security for RAM programs is minimal and does not grow with the running time of the program.

RANDOM ACCESS MACHINE IN SECURE MULTI-PARTY COMPUTATION

by

Zhangxiang Hu

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 3, 2015
Commencement June 2016

Master of Science thesis of Zhangxiang Hu presented on June 3, 2015.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Zhangxiang Hu, Author

ACKNOWLEDGEMENTS

Great thanks go to my advisor Mike Rosulek for his advising during my studies, this thesis would never possible without him. He inspired me a lot on ethics both in academia and research, and gave a lot of insightful comments on technical writings in cryptography. It is my honor to be his student.

I am grateful to the my thesis committee members, Attila Altay Yavuz, Amir Nayyeri, Rakesh Bobba and Mike Pavol for their helpful feedback on this thesis. Also thanks to my collaborators and co-authors, Arash Afshar and Payman Mohassel.

Finally but most importantly, I would like to thank my family for their great support in my life, especially my wife Xueni Guo who undertook the unpredictable journey to USA, she means the whole world to me.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

Table Page

LIST OF APPENDIX TABLES

Table Page

Chapter 1: Introduction

The concept of secure two-party computation in the presence of semi-honest adversary was introduced by Yao in [55]. It allows two parties to jointly perform computations on their private inputs without leaking any information about their input beyond what is deducible from the output of computation. Namely, there are two parties Alice and Bob with their respective private input x and y, and they wish to cooperatively evaluate a polynomial-time computable function $f(x, y) = (f_1(x, y), f_2(x, y))$ such that Alice receives output $f_1(x, y)$ and Bob receives output $f_2(x, y)$. Besides the output, they should learn no useful information about the other parties' private input. A more general case is multi-party computation (MPC) rather than two parties. There are many situations in the real world that can take advantage of multi-party computation. For example, in electronic voting, voters want to choose a new president but are unwilling to reveal who they voted for. Another common situation is an auction, all parties want to find out the highest bidder but they do not want let others know their pricing strategy.

A naive way to achieve this is to find a trusted third party to gather all inputs, do the computation and announce the results to all parties. In the real world, a trusted third party can be a lawyer, a well-known company or government. However, sometimes it is hard to find a trusted third party, so we want a good "strategy" that can simulate the trusted party and perform the computation. Normally, a good "strategy" must achieve some security properties such as correctness, privacy, authenticity etc.

In cryptography, such strategy is called protocol. It implements the functionality of a trusted third party: performing the computation while still keeping the privacy of each party's input. Originating from the works [55, 16], there are significant improvements in designing and implementing a practical secure computation protocol. These techniques are mostly restricted to functions represented as boolean or arithmetic circuits. However, the conversion to circuit may lead to a huge blowup both in circuit size and running time. In addition, in the real world, the majority of applications we encounter in practice are more efficiently captured using random-access machine (RAM) programs that allow constant-time memory lookup. For example, to search an element in a large database,

efficient programs such as binary search can run in sub-linear time in the size of data. To convert it to a circuit, there are known polynomial transformations between RAM programs, Turing Machines and circuits. Given a RAM program with running time T, we can convert it to a Turing machine with running time $O(T^3)$ [10]; and such Turing machine can be transformed to a circuit of size $O(T^3 \log T)$ [44]. Therefore, by converting a RAM program to a circuit, it incurs an expensive time lose.

An variant of RAM program is oblivious RAM (ORAM). ORAM is a secure implementation of RAM program computation. It was first introduced by Goldreich and Ostrovsky [18] to protect software from illegitimate duplication. Different from the regular RAM program, an ORAM program can hide the information of both data and access pattern. In section 2.4, we present more details about ORAM programs.

1.1 Security Concerns

In this section, we introduce some fundamental notions in cryptography.

1.1.1 Adversary

To model security, we first need to model our adversary. That is, if a party is corrupted by an adversary, what can the adversary do when performing the computation? The security definition depends on the assumptions of adversary's ability. In general, there are two main adversary models that have been considered:

- **Semi-honest:** Also called "passive". In this model, the adversary follows the protocol correctly as an honest party, but tries to learn some unauthorized information from the transcript of messages it received during the computation.

- **Malicious:** Also called "active". In contrast to the semi-honest model, the malicious adversary can deviate arbitrarily from the protocol during the computation. Usually, we prefer our protocol to protect against malicious adversaries. In this case, it guarantees that an attack to the protocol can succeed only with negligible probability. However, it also requires more cost and has less efficiency than semi-honest model.

Another important distinction of adversary in MPC is between adaptive and non-adaptive. An adaptive adversary allows to choose parties to be corrupted throughout the computation while a non-adaptive adversary allows to control arbitrary but fixed set of corrupted parties.

1.1.2 Ideal Functionality and Protocol

Functionality defines what a trusted third party can do for the computation. We usually refer it as in the ideal world, it describes what kind of computation task we want to achieve. The ideal functionality repeatedly receives inputs from all parties and sends back appropriate output values to them. It guarantees that all parties' outputs have the expected properties with respect to their inputs.

A functionality is called *reactive* if new input values are received and new output values are generated throughout the computation. [1] Otherwise a functionality is called *non-reactive* if it is just waiting for inputs from all parties and sends corresponding outputs back to them.

A protocol π for a functionality $\mathcal{F}$ is an algorithm that consists of all parties exchanging information with each other to compute the same result as the functionality. A protocol with n participants is called an n-party protocol. We usually refer to protocols in the real world, it implements the task of a ideal functionality.

1.1.3 Security Analysis

To analyze the security of the protocol, we usually use *real/ideal world* paradigm, introduced by Goldreich, Micali, and Wigderson [16]. In this paradigm, the real world has all parties engage in a protocol while the ideal world has all parties simply send their input to a trusted third party (the functionality) who performs the computation. We say that the protocol securely implements the functionality if the real world is as secure as the ideal world. Namely, let $\mathcal{S}$ be a simulator (usually is an algorithm) that takes whatever the adversary can see in the ideal world, $\mathcal{S}$ simulates the view that the adversary sees in the real world. We use $\mathsf{view}_{\mathcal{S}}$ to denote the simulated real world view described by simulator $\mathcal{S}$, and use $\mathsf{view}_{\mathcal{R}}$ to denote the real view in the real world. If one can not

[1] Even if the new inputs are chosen based on previous outputs.

efficiently distinguish between $\mathsf{view}_{\mathcal{S}}$ and $\mathsf{view}_{\mathcal{R}}$, then the adversary learns nothing new from the protocol execution. In other words, if an adversary can attack the real world interaction, it can also attack the ideal world interaction and achieve the same effect.

In multi-party computation, all parties' inputs are chosen independent of other parties' input. In the ideal world, the trusted third party takes all parties' input, performs the computation and sends the corresponding result to each party. Whatever can be seen in the real world for each party must be also learned only from party's output in the ideal world. Both views should be computational indistinguishable. We will discuss more about indistinguishability in section 2.1.

The security described above is called *standalone security*. However, it only address the case that a single protocol execution is considered and it does not guarantee the security when a protocol instance may run concurrently with other protocols. Thus, we adopt the framework by Canetti [8] which is called *Universally Composable Security* or *UC-Security*. Here we only give a brief overview, and refer the reader to [8] for the full definition.

Environments. This framework uses the notion of an *external environment* to model contexts in which a protocol might be asked to execute. The environment interacts with all parties throughout the execution and tries to help the adversary. It sends some inputs to parties and receives outputs from them. Indeed, environment interacts with the protocol execution twice.

- First, it sends arbitrary inputs to the parties and to the adversary, and then it receives the outputs from the parties and the adversary.
- Second, the environment outputs a single bit, which indicates whether the environment thinks that it has interacted with the real protocol or with the ideal functionality.

Dummy protocol. An important protocol in this UC framework is called the *dummy protocol*, denoted by π_{dummy}, it prescribes the same behavior for each party as they would execute protocol π in the real world.

Execution. Let $\mathcal{F}$ be a functionality, π be a protocol, $\mathcal{A}$ be an adversary, and $\mathcal{Z}$ be an environment, we define $\mathsf{EXEC}(\mathcal{F}, \pi, \mathcal{A}, \mathcal{Z}, k)$ to be the random variable of the environment $\mathcal{Z}$'s output, it can be only 0 or 1. Here k is the secure parameter.

Definition 1 ([48]). *Let $\mathcal{F}$ and $\mathcal{G}$ be functionalities, and π be a protocol, we say π is a secure realization of $\mathcal{F}$ in the $\mathcal{G}$-hybrid setting if for all PPT real-world adversaries $\mathcal{A}$, there exists another PPT adversary (called a simulator) $\mathcal{S}$ such that for all PPT environments $\mathcal{Z}$, we have*

$$Pr[\mathsf{EXEC}(\mathcal{G}, \pi, \mathcal{A}, \mathcal{Z}, k) = 1] \approx Pr[\mathsf{EXEC}(\mathcal{F}, \pi_{\mathsf{dummy}}, \mathcal{S}, \mathcal{Z}, k) = 1]$$

Here we refer π and $\mathcal{G}$ as in the real world, while refer $\pi_{\mathsf{dummy}}$and $\mathcal{F}$ as in the ideal world. This definition implies that the real world is as secure as the ideal world. In other words, whatever an adversary can do to attack against π in the real world can also be done in the ideal world.

Now we come up with the fundamental result in the UC framework as follows:

Theorem 2 ([8]). *If π is a secure realization of $\mathcal{F}$ in the $\mathcal{G}$-hybrid world, and ρ is a secure realization of $\mathcal{G}$ in the $\mathcal{H}$-hybrid world, then π^ρ is a secure realization of $\mathcal{F}$ in the $\mathcal{H}$-hybrid world, where π^ρ is the protocol π in which each external interface to an instance of $\mathcal{G}$ is replaced with an instance of ρ.*

In other words, if $\mathcal{G}$ is a functionality in protocol π, we can always replace $\mathcal{G}$ with its secure protocol ρ and vice versa.

1.2 Our contribution

In this section, we present a brief overview about what are achieved in this thesis.

1.2.1 ORAM Application in Zero-knowledge Proof

In chapter 3, we present a practical application of ORAM in zero-knowledge proof. We describe a zero-knowledge proof system in which a prover holds a large dataset M and can repeatedly prove NP relations about that dataset. Each proof requires only constant number of rounds of interaction and has sublinear amortized cost in $|M|$. In addition,

the storage requirement between proofs for the verifier is constant. Our construction combines an Oblivious RAM and garbled circuits, but without using cryptographic operations inside the garbled circuits as in current garbled-RAM constructions, and thus has a high efficiency in communication/computation complexity.

This work represents joint work with Mike Rosulek from Oregon State University and Payman Mohassel from Yahoo Labs, and will appear in conference CRYPTO 2015.

1.2.2 RAM Program with Malicious Security

In chapter 4, we present the first practical protocols for evaluating RAM programs with security against malicious adversaries. Our RAM protocols achieve ratios matching the state of the art for circuit-based 2PC, the extra overhead of obtaining malicious security for RAM programs (beyond what is needed for circuits) is minimal and does not grow with the running time of the program. We introduce two protocols, which use different approaches for reusing wire labels. The first protocol uses ideas from the LEGO paradigm for 2PC [41, 13] and the second protocol directly reuses wire labels without soldering. We also show how to incorporate the input recovery technique of [32] for reducing the number of circuits by a factor of three.

This work represents joint work with Arash Afshar from University of Calgary, Payman Mohassel from Yahoo Labs and Mike Rosulek from Oregon State University, and is appeared in conference EUROCRYPT 2015 [1].

Chapter 2: Preliminary

In this section, we introduce some basic notations and preliminary notions and definitions that used throughout this work.

2.1 Basic Notation

Let $\mathbb{N}$ denotes the set of natural numbers $\{0, 1, 2, \ldots\}$ and let $k \in \mathbb{N}$ be the security parameter. We say a function $\epsilon : \mathbb{N} \to [0, 1]$ is negligible if for any polynomial p, there exists a large enough k' such that for all $k > k'$, $\epsilon(k) < 1/p(k)$. We say that a probability $\mathsf{Pr}(k)$ is overwhelming if $1 - \mathsf{Pr}(k)$ is a negligible function in k.

Let $\mathcal{D}_0$ and $\mathcal{D}_1$ be two discrete probability distributions. A **distinguisher** $\mathcal{A}$ is a deterministic algorithm that outputs either 0 or 1. We define the bias of $\mathcal{A}$ in distinguishing $\mathcal{D}_0$ from $\mathcal{D}_1$ as:

$$\text{bias}(\mathcal{A}, \mathcal{D}_0, \mathcal{D}_1) = \Big| \Pr[x \leftarrow \mathcal{D}_0 : \mathcal{A}(x) = 1] - \Pr[x \leftarrow \mathcal{D}_1 : \mathcal{A}(x) = 1] \Big|$$

and define the statistical distance between distributions $\mathcal{D}_0$ and $\mathcal{D}_1$ as:

$$\Delta(\mathcal{D}_0, \mathcal{D}_1) = \max_{\mathcal{A}} \text{bias}(\mathcal{A}, \mathcal{D}_0, \mathcal{D}_1)$$

We say that two distributions $\mathcal{D}_0$ and $\mathcal{D}_1$ are computationally indistinguishable if for all polynomial-time distinguisher $\mathcal{A}$, the bias of $\mathcal{A}$ in distinguishing $\mathcal{D}_0$ from $\mathcal{D}_1$:

$$\Pr[x \leftarrow \mathcal{D}_0 : \mathcal{A}(x) = 1] - \Pr[x \leftarrow \mathcal{D}_1 : \mathcal{A}(x) = 1]$$

is negligible.

For an integer n, we define $[n] = \{1, 2, \ldots, n\}$.

2.2 Hash Function and Strongly Universal Hashing

A hash function $h : \{0,1\}^* \mapsto \{0,1\}^n$ is any function that takes arbitrary-length input and has fixed length output. We say x, x' with $x \neq x'$ are a collision under h if $h(x) = h(x')$. And h is collision resistant if for all PPT algorithms, the probability that we can find a collision in h is negligible.

A hash function family $\mathcal{H}$ is a set of hash functions where each hash function in $\mathcal{H}$ has the same output length. We also require the collision resistant property, in a hash function h which is chosen uniform randomly from the family. Notice that the difficulty of finding collisions now rests in the random choice of functions. Even an adversary can know every fact about $\mathcal{H}$, it still doesnt know which $h \in \mathcal{H}$ it is going to be challenged to find a collision.

Let $\mathcal{H}$ be a hash function family and each function $h \in \mathcal{H}$ has the form $h : A \mapsto B$, then $\mathcal{H}$ is strongly universal if for all distinct $a, a' \in A$ and all (possibly equal) $b, b' \in B$,

$$\Pr_{h \leftarrow \mathcal{H}}[h(a) = b \wedge h(a') = b'] = 1/|B|^2$$

One choice of strongly universal families is linear functions. In a finite field $\mathbb{F}$, define function $f_{a,b}(x) \mapsto ax + b$, $a, b \in \mathbb{F}$, then the class of functions $\{h_{a,b} | a, b \in \mathbb{F}\}$ is a strongly universal function family.

2.3 Garbled Circuit and Garbling Scheme

2.3.1 Garbled Circuit

The first solution for secure two-party computation was also introduced by Yao [56]. The idea is by using the technique *garbled circuits*. Denote party P_1 as the sender with input x and P_2 as the receiver with input y. Let f to be the function that they wish to compute. For simplicity, we assume $f_1(x, y) = f_2(x, y)$. The first step is to express f as a boolean circuits C_f. Then P_1 garble the circuit and sends it to P_2. Such circuit reveals nothing since it is encrypted. P_2 evaluates the circuit based on input (x, y) to obtain the output $f(x, y)$ and then announce the results. During the evaluation, P_2 should only learn the output $f(x, y)$. Other messages that P_1 and P_2 receive must leak no information.

Consider the evaluation of a circuit, a circuit is computed gate by gate, from the input wire to the output wire. Consider a fan-in to 2, and fan-out to 1 gate g in figure 2.1, the gate input wires w_1, w_2 has value $\alpha, \beta \in \{0,1\}$, then we can obtain the value $g(\alpha, \beta)$ of gate output wire w_3. We keep doing this until all gates are evaluated. The value of circuit outputs wires are the output of circuit.

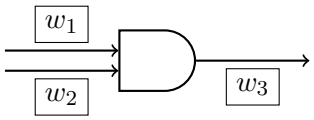

Figure 2.1: And gate

Next we show how to garble a circuit and how to evaluate it. Consider the same AND gate g with input wires w_1, w_2 and output wire w_3. The general idea of Yao's protocol is to generate two keys(also called wire labels) for each wire, and it does not reveal anything about the original wire values. For the AND gate g, we generate two random cryptographic keys k_i^0 and k_i^1 for wire w_i. Here k_i^0 represents the value 0 and k_i^1 represents the value 1. Notice that if one receives $k_i^\sigma, \sigma \in \{0,1\}$, it can not tell σ is 0 or 1 since the keys have identical distribution. Now we have k_1^0, k_1^1 for input wire w_1, k_2^0, k_2^1 for input wire w_2 and k_3^0, k_3^1 for output wire w_3. Next we have the output value k_3^0, k_3^1 encrypted under the corresponding keys from the input wires. See the garbled table for AND gate g:

Input wire w_1	Input wire w_2	Output wire w_3	Encryption table
k_1^0	k_2^0	k_3^0	$\text{Enc}_{k_1^0}(\text{Enc}_{k_2^0}(k_3^0))$
k_1^0	k_2^1	k_3^0	$\text{Enc}_{k_1^0}(\text{Enc}_{k_2^1}(k_3^0))$
k_1^1	k_2^0	k_3^0	$\text{Enc}_{k_1^1}(\text{Enc}_{k_2^0}(k_3^0))$
k_1^1	k_2^1	k_3^1	$\text{Enc}_{k_1^1}(\text{Enc}_{k_2^1}(k_3^1))$

Table 2.1: Garbled table for AND gate

Here **Enc** is a private-key encryption scheme that is secure under *chosen plaintext attacks*. An example for **Enc** suggested in [34] is:

$$\mathbf{Enc}_k(x) = \langle r, f_k(r) \oplus x0^n \rangle$$

- On input (x_0, x_1) from party P_1, and input $\sigma \in \{0,1\}$ from P_2, sends delayed output x_σ to P_2 and $\perp$ to P_1.

Figure 2.2: 1-out-of-2 oblivious transfer functionality $\mathcal{F}_{\mathsf{ot}}$

where $x \in \{0,1\}^n, r \in_R \{0,1\}^n$, $f_k : \{0,1\}^n \to \{0,1\}^{2n}$ for $k \in \{0,1\}^n$ is a pseudorandom function. Now if we have two input wire keys, we can obtain the corresponding output wire key by decrypting the four entries in the garbled table. It is guaranteed that only one ciphertext can be decrypted correctly. Meanwhile, the original value of wire w_i is still hidden. This is because the fact that the evaluator does not know the key is associated with zero or one.

To securely evaluate function f, P_1 generates the circuit C_f and sends her wire labels to P_2 according to her input x. Now if P_2 knows his corresponding wire labels, then P_2 can evaluate the circuit. Notice that P_1 has both wire labels and P_2 has his own input value y. However, P_1 does not know which wire label he should send to P_2 since he can not know y. Thus, we need a technique to let P_2 receives his corresponding wire labels.

2.3.2 Oblivious Transfer

To solve the above problem, Robin introduced oblivious transfer (OT) in [46]. Generally speaking, a k-out-of-n OT is a protocol in which the sender has a list of n messages $\{m_1, \cdots m_n\}$, the receiver receives k of them without learning anything about other $n-k$ messages. Also, the sender would have no idea about which k messages the receiver has received. Throughout this work, we will use 1-out-of-2 OT as defined in figure 2.2. In standard 1-out-of-2 OT, party P_1 inputs two messages (x_0, x_1), and party P_2 has input σ. After running oblivious transfer, P receives x_σ.

Committing oblivious transfer The definition of committing oblivious transfer was first given by Kiraz and Schoenmakers [27]. It works the same as a standard OT except that the "committing" aspect allows party P_2 to reveal (x_0, x_1) at a later time. The ideal functionality $\mathcal{F}_{\mathsf{otc}}$ is defined in Figure 2.3.

- Initialization: $\mathcal{F}_{\mathsf{otc}}$ takes private input (x_0, x_1, id) from party P_1 and the private input $\sigma \in \{0,1\}$ from party P_2, then stores (x_0, x_1, id, σ) internally and output COMMITTED.
- Transfer: On command $(\text{TRANSFER}, id)$ from P_1, $\mathcal{F}_{\mathsf{otc}}$ sends $(\text{TRANSFERRED}, x_\sigma, id)$ to P_2.
- Open: On command (OPEN, id) from P_1, $\mathcal{F}_{\mathsf{otc}}$ sends $(\text{OPENED}, x_0, x_1, id)$ to P_2.

Figure 2.3: Ideal functionality $\mathcal{F}_{\mathsf{otc}}$ for committing oblivious transfer.

2.3.3 Garbling Scheme

In section 2.3.1, we have a brief overview about how to use garbled circuit to implement two party secure computation. We have to interpret f as a circuit, and then garble such circuit and evaluate it. In this section, we present a high level abstraction of the garbled circuit from [5]. The abstraction formalize garbled circuits into a primitive which they call garbling scheme. Bellare *et al.* abstract versatile syntax and security definitions such that any garbled circuit satisfies their abstraction.

2.3.3.1 Syntax

A garbling scheme can be defined as a five-tuple algorithm: $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$. Denote $f : \{0,1\}^n \to \{0,1\}^m$ to be the function that we want to evaluate, let x to be the original input and $y = f(x)$ to be the final output. Then Gb is a randomized algorithm that transform f into a new triple of functions (F, e, d). En is an algorithm that takes (e, x) as the input and outputs garbled input $X = \mathsf{En}(e, x)$. Ev takes input (F, X) and outputs the garbled output $Y = \mathsf{Ev}(F, X)$. De is an algorithm that transforms garbled output Y to the final output $y = \mathsf{De}(d, Y)$ and we have $\mathsf{De}(d, Y) = \mathsf{ev}(f, x)$. See figure 2.4 from [5]. Specifically, for a garbled circuit, we say $\mathcal{G}$ is a circuit garbling scheme if ev interprets f as a circuit.

Definition 3. *A garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ satisfies correctness if for all f and for all x, it holds the condition that*

$$\mathsf{De}(d, \mathsf{Ev}(F, \mathsf{En}(e, x))) = \mathsf{ev}(f, x)$$

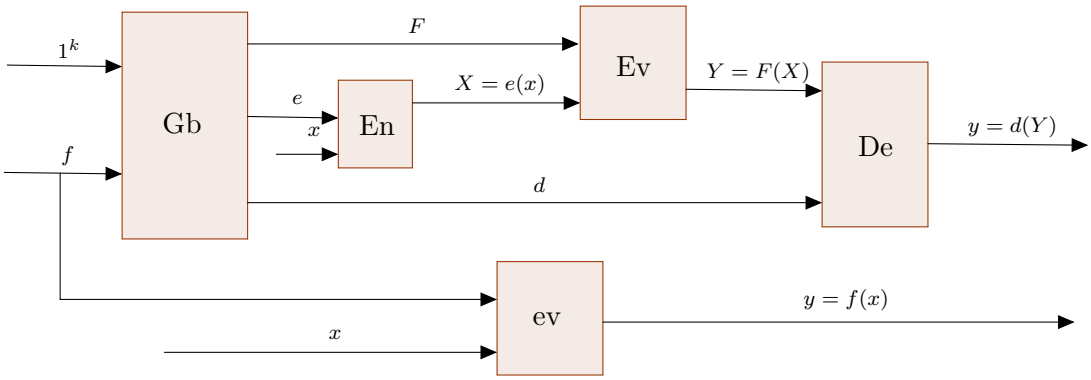

Figure 2.4: Description of garbling scheme

2.3.3.2 Security

In [5], the security of a garbling scheme can be abstracted into three properties: *Privacy*, *Obliviousness* and *Authenticity*. Let Φ be the side information function that represents the information we expect to reveal for the garbling scheme.

1. **Privacy** For a simulation based privacy, it requires that whatever can be learned from (F, X, d) can also be learned by just knowing the final output y and side information function Φ. More formally, we define privacy as follows.

   Definition 4. *A garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ satisfies privacy property if for every function f, input x, side information function Φ, and for all PPT adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that the following difference:*

   $$\left| \Pr[\mathcal{A}(\mathcal{S}(1^k, y, \Phi(f)) = 1] - \Pr[\mathcal{A}(F, X, d) = 1] : (F, e, d) = \mathsf{Gb}(1^k, f), X = \mathsf{En}(e, x) \right|$$

   is negligible in k.

2. **Obliviousness**. Informally, obliviousness means when given $Y = \mathsf{En}(F, X)$ but not d, it does not leak any information about f, x, y beyond $\Phi(f)$.

   Definition 5. *A garbling scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ satisfies obliviousness property if for every function f, input x, side information function Φ, and for all*

PPT adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ such that the following difference:

$$\Big|\Pr[\mathcal{A}(\mathcal{S}(1^k,\Phi(f))=1]-\Pr[\mathcal{A}(F,X)=1]:(F,e,d)=\mathsf{Gb}(1^k,f),X=\mathsf{En}(e,x)\Big|$$

is negligible in k.

3. **Authenticity**. Roughly speaking, authenticity means when given F and X, the adversary can not forge a valid Y such that $Y \neq \mathsf{Ev}(F,X)$ and $d(Y) \neq \perp$.

   Definition 6. *A garbling scheme $\mathcal{G} = (\mathsf{Gb},\mathsf{En},\mathsf{De},\mathsf{Ev},\mathsf{ev})$ satisfies authenticity property if for every function f, input x, side information function Φ, and for all PPT adversary $\mathcal{A}$, the following probability:*

$$\Big|\Pr[\mathsf{De}(d,Y)\neq\perp, Y\neq\mathsf{Ev}(F,X):(F,e,d)=\mathsf{Gb}(1^k,f),X=\mathsf{En}(e,x),Y=\mathcal{A}(F,X)]\Big|$$

   is negligible in k.

2.4 Oblivious RAM

Oblivious RAM (ORAM) programs were first introduced by Goldreich and Ostrovsky [18] to protect software from illegitimate duplication. This technique allows the CPU to hide all information about a RAM program, both the memory contents it accessed and its corresponding access patterns. In particular, if an adversary is eavesdropping the communication between the CPU and the memory, it can not tell what data the CPU is accessing and what memory the CPU is really trying to access. Theses information should look random to the adversary. The probability distribution of the access pattern must depend only on the input length rather than the input itself.

A more general application of ORAM is between a client with a limited memory and an untrusted server with a large memory. The client wants to store a large dataset on the server side and operate the data while still maintaining the privacy. An intuitive way is to encrypt the dataset before storing it to server. However, it still leaks the access pattern which contains significant sensitive information. Therefore, we use oblivious RAM program here to hide the information of both data and access pattern. Notice that an adversary can observe the physical locations that a client has accessed, but it

reveals nothing about the virtual memory and the physical access sequence should look random to the adversary.

In general, a ORAM program can be implemented as a circuit and running a ORAM program is identified as evaluating such corresponding deterministic *next-instruction* circuit Π. Assume that there is a ORAM program Π with its logical memory M. First we invoke a initialize function $\mathsf{Initialize}$ to encode the logical memory M into the physical memory array $\widehat{M}$:

$$(\widehat{M}, st) \leftarrow \mathsf{Initialize}(1^k, M)$$

At each evaluation step, Π checks its current ORAM state st, takes input of Σ and $block$ where Σ is external input and $block$ is the physical memory block data, outputs an instruction $inst$, next ORAM state and the corresponding data that is used for updating the memory block. The syntax of next-instruction circuit Π of a ORAM program is defined as follow:

$$(inst, st, block) \leftarrow \Pi(st, \Sigma, block)$$

The $inst$ variable can only have one of the three forms: (READ, i), (WRITE, i), or (HALT, z). (READ, i) means the ORAM program to read data from block $\widehat{M}[i]$ and assign it to $block$ while (WRITE, i) means the ORAM program to write data $block$ to the memory block $\widehat{M}[i]$. (HALT, z) indicates the ORAM program to terminate and output the final result z.

More formally, let $\mathcal{I}$ denote the access sequence and $r \leftarrow \{0,1\}^k$ be the external random input to the ORAM program, an execution of an ORAM program Π on input x with logical memory M is defined as follows:

$\underline{\text{RAMEval}(\Pi, M, x)}$
$\mathcal{I} := \emptyset$, $(\widehat{M}, st) \leftarrow \mathsf{Initialize}(1^k, M)$
$(inst, st, block) := \Pi(st, x, \perp)$
do until inst has the form (HALT, z):
    $block :=$ [if $inst = (\text{READ}, id)$ then $\widehat{M}[id]$ else $\perp$]
    $r \leftarrow \{0,1\}^k$, $(inst, st, block) := \Pi(st, r, block)$
    if $inst = (\text{WRITE}, id)$ then $\widehat{M}[id] := block$
    $\mathcal{I} := \mathcal{I} \| inst$
output z

The security definition of an ORAM program Π requires that the memory access sequence $\mathcal{I}$ does not leak any information about the data set M or the input x. In other words, let $\mathcal{I}(\Pi, M, x)$ be the random variable denotes the sequence of values taken by the *inst* variable in $\text{RAMEval}(\Pi, M, x)$, then there exists a simulator $\mathcal{S}$ such that, for all x and initially empty $\widehat{M}$, the output $\mathcal{S}(1^k, z)$ is indistinguishable from $\mathcal{I}(\Pi, M, x)$ where z is the final output of the RAM program on inputs x.

Definition 7. *We say that* Π *is a* **secure ORAM** *if there exists an efficient simulator* $\mathcal{S}$ *such that, for all* M*, all* $(\widehat{M}, st) \leftarrow \mathsf{Initialize}(1^k, M)$*, all* x *and* z *such that* $z = RAMEval(\Pi, M, x)$ *and for all PPT* $\mathcal{A}$*, the following difference:*

$$\left| \Pr[\mathcal{A}(\mathcal{S}(1^k, z) = 1] - \Pr_r[\mathcal{A}(\mathcal{I}(\Pi, M, x)) = 1] \right|$$

is negligible in k*.*

2.5 Commitment and Zero-knowledge proof

2.5.1 Commitment

The notion of commitment scheme is one of the most fundamental tool in cryptography. It allows one party to create a commitment c to a secret message m while still hiding the message from other parties, but later the party can choose to open the commitment and disclose m. It requires that the commitment must *binding*, which means once the commitment c is made, it is impossible to find another message m' such that m' and m has the same commitment c. The commitment functionality $\mathcal{F}_{\mathsf{com}}$ is defined in figure 2.5.

In addition to a standard commitment functionality $\mathcal{F}_{\mathsf{com}}$, an XOR-homomorphic commitment scheme allows party P_1 to disclose the XOR of two or more committed messages while keeps hiding the individual messages. The XOR-homomorphic commitment functionality $\mathcal{F}_{\mathsf{xcom}}$ is defined in figure 2.6.

Let $\mathcal{M}$ denote the space of valid messages and the functionality is parameterized by an integer k.

- Commit: On input (COMMIT, m) from party P_1 with $m \in \mathcal{M}$ and $m \in \{0,1\}^k$, if there is no value m already stored in memory, then $\mathcal{F}_{\mathsf{com}}$ stores m internally and outputs COMMITTED to party P_2.
- Open: On input OPEN from P_1, if value m exists in memory, then $\mathcal{F}_{\mathsf{com}}$ outputs (OPENED, m) to party P_2.

Figure 2.5: Ideal functionality $\mathcal{F}_{\mathsf{com}}$ for commitment

The functionality is initialized with internal value $i = 1$. It then repeatedly responds to commands as follows:

- On input (COMMIT, m) from P_1, store (i, m) internally, set $i := i + 1$ and output $(\text{COMMITTED}, i)$ to both parties.
- On input (open, S) from P_1, where S is a set of integers, for each $i \in S$ find (i, m_i) in memory. If for some i, no such m_i exists, send $\perp$ to P_2. Otherwise, send $(\mathit{open}, S, \bigoplus_{i \in S} m_i)$ to P_2.

Figure 2.6: XOR-homomorphic commitment functionality $\mathcal{F}_{\mathsf{xcom}}$.

2.5.2 Zero-knowledge Proof

In zero-knowledge proof, a prover wants to convince a verifier that some NP statement x is true by using a valid witness w, and the verifier learns nothing except the validity of the statement. Typically, for any language $\mathcal{L} \in \mathsf{NP}$ with some binary relation $\mathcal{R}_{\mathcal{L}}$, for all valid instances $x \in \mathcal{L}$, there exists a string w such that $\mathcal{R}_{\mathcal{L}}(x, w) = 1$. Otherwise, if $x \notin \mathcal{L}$, then for all string w we have $\mathcal{R}_{\mathcal{L}}(x, w) = 0$. The ideal functionality $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ is defined in figure 2.7.

A zero-knowledge proof protocol must satisfy the following three properties:

- Completeness The prover can always convince the the verifier of true statements.
- Soundness False statement will not be accepted by the verifier, and the dishonest prover will be caught cheating with overwhelming probability.
- Zero-knowledge Roughly speaking, zero-knowledge means the verifier learns noth-

$\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ is parametrized by a relation $\mathcal{R}$. It involves two parties: a prover P and a verifier V.

- Setup: On input (INIT, M) from P, if no previous INIT command has been given, then $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ stores M internally.
- Proof: On input $(\text{PROVE}, sid, x, w)$ from P, if $\mathcal{R}(M, x, w) = 1$, output (ACCEPT, sid, x) to V.

Figure 2.7: Ideal functionality $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ for zero-knowledge proofs of NP-relation $\mathcal{R}$

ing during the proof except that the statement is true. Formally, we require that there exists a efficient simulator $\mathcal{S}$ such that, for all PPT verifier, $\mathcal{S}$ can simulate all interactions of verifier with the prover.

Chapter 3: Zero-Knowledge Proofs by using Oblivious RAM program

3.1 Introduction

Since zero-knowledge proof plays such a important role in cryptography, there are many fundamental results show that for all languages in NP, there are zero-knowledge proofs, [12, 17, 23, 45]. However, it shows that they are too inefficient to be used in practical applications in the real world.

Besides these protocols of mainly theoretical interested, the recent work [49, 11, 7, 20] also introduce various protocols that are efficient enough for real world use. But all of their constructions were practical only for proving statements about certain algebraic structures such as proving knowledge of and relations for discrete logarithms, RSA public keys, and bilinear equations.

The recent work [24] combines zero-knowledge proof and garbled circuit and proposes a new approach that is suitable for general-purpose statements represented as boolean circuits. The general idea is to interpret the binary relation $\mathcal{R}_{\mathcal{L}}$ as a function $f_{\mathcal{R}}$ such that for some statement x, $f_{\mathcal{R}}(w) = 1$ if w is a valid witness of x (which means $\mathcal{R}_{\mathcal{L}}(x, w) = 1$) and otherwise $f_{\mathcal{R}}(w) = 0$. Then we can use Yao's garbled circuit technique to garble $f_{\mathcal{R}}$ so both parties can evaluate the circuit. The security property of garbled circuit will guarantee the completeness, soundness and zero-knowledge requirements in zero-knowledge proofs.

In addition, a key observation here is that only the prover has input w to function $f_{\mathcal{R}}$ while the verifier does not, which means there is no security needed for the verifier. Thus, some very efficient garbling schemes can be applied here. [29] shows that the number of ciphertexts we need for each gate in a garbled circuit can be reduced to $0, 1$, or 2. It significantly reduce the communication complexity and computational complexity, achieves a very practical use in the application world.

3.1.1 Our results

Garbled circuit-based zero-knowledge proofs are very efficient and with the state-of-art circuit garbling technique [29, 30], zero-knowledge proofs can scale to statements of billions of gates. However, it still has limitations in some scenario. For example, a prover commits to a dataset S and wants to prove membership and non-membership statements x ($x \in S$ and $x \notin S$). Since garbled circuit is one time, we must construct a new circuit for each statement. Otherwise, the authenticity property of garbled circuit will not be guaranteed. This approach is so cumbersome and it becomes much worse when the dataset S is very large.

In our work, we introduce a new approach for zero-knowledge proof to address the problem that described above. We combines oblivious RAM and garbled circuit but without using cryptographic operations inside the garbled circuits as in current garbled-RAM constructions. It requires only a constant number of rounds of interaction. Also, in general we do not have to access the whole dataset, we only need the memory blocks that will be used in ORAM program. Therefore, the size of circuit ORAM is much smaller than [24].

Let Π be a oblivious RAM program and M is its corresponding memory blocks. For any statement x and witness w of the form $\exists w : R(M, x, w) = 1$, our solution is constant-round, and incurs online computation and communication cost that is linear in the running time of the RAM program, competitive with the best semi-honest 2PC for RAM programs [19], and hence sublinear in $|M|$ for many applications of interest. Sublinear-time 2PC is not possible in general when expressing the NP relation as a boolean circuit. Furthermore, in our protocol the verifier maintains only constant storage space between multiple proofs.

3.2 Preliminaries

3.2.1 Authenticated Array

An authenticated array allows one party to access all stored data and control over all modifications in the array, as the functionality $\mathcal{F}_{\mathsf{Aut}}$ is described in figure 3.1. A naive implementation is to let party to generate the array in the local storage. However, when the party V has minimal memory and it wants to store the array in some untrusted

party, then V need another way to guarantee the authenticity of such array. A simple solution is to use an *authenticated Merkle-tree*, with V storing only the root of the tree.

- Initialization: On input (INIT, N) from party V where $N \in \mathbb{N}$, $\mathcal{F}_{\mathsf{Aut}}$ initialize an array T of size N. For each $T[i]$, $i \in \{1, \ldots, N\}$, set $T[i] = 0$.
- Update: On input $(\text{UPDATE}, id, data)$ from party V, set $T[id] = data$ and output $(\text{UPDATED}, id, data)$ to both parties.
- Open: On input (ACCESS, id) from party V, where $id \in \{1, \ldots, N\}$, send $(\text{ACCESSED}, id, T[id])$ to V.

Figure 3.1: Ideal functionality $\mathcal{F}_{\mathsf{Aut}}$ for authenticated array access.

3.2.2 Committing Private Function Evaluation

Private function evaluation (PFE) takes input a function h from a sender and input x from a receiver, and gives output $h(x)$ to the receiver. We define and use a committing variant of PFE in which the sender can later reveal the h that was used. The formal description is given in Figure 3.2. In addition, we also require that committing PFE supports a strongly universal class $\mathcal{H}$ of functions.

$\mathcal{F}_{\mathsf{cpfe}}$ is parametrized by a class of functions $\mathcal{H}$, with each $h \in \mathcal{H}$ having a common domain A.

- Evaluation: On input $h \in \mathcal{H}$ from party V and input $x \in A$ from party P, give output $h(x)$ to party P. Remember h internally.
- Open: On input OPEN from party V, give output h to party P.

Figure 3.2: Ideal functionality $\mathcal{F}_{\mathsf{cpfe}}$ for committing private function evaluation.

A feasible approach to implement $\mathcal{F}_{\mathsf{cpfe}}$ is to use *oblivious linear function evaluation* (OLFE) [54]. Let $\mathbb{F}$ be a finite field, then the class of functions of the form $x \mapsto ax + b$ is strongly universal (with $a, b \in \mathbb{F}$). OLFE takes input a, b from the sender and x from the receiver, and sends the output $ax + b$ to the receiver.

3.2.3 Garbling Scheme

The garbling scheme we use in this work is slightly different from the standard garbling scheme in section 2.3.3. Recall that in the standard garbling scheme, we use X and Y to represent the real garbled input and garbled output respectively. When f is a circuit, to garble the circuit, we use E and D to represent the descriptions of input wire labels and output wire labels. So X is a subset of E and Y is a subset of D. [1] In other words, let W be a $m \times 2$ array that represent a set of wire labels on m wires, For each wire i, $W[i,0] \in \{0,1\}^k$ and $W[i,1] \in \{0,1\}^k$ are two wire labels that encode FALSE and TRUE, respectively. For a truth value x, the corresponding wire labels are defined as $W|_x = (W[1,x_1], \ldots, W[m,x_m])$. Therefore, for genuine input x and output y, we have $X = E|_x$ and $Y = D|_y$.

Also, we have different security requirements in our protocol. Specifically, in zero-knowledge proof, we require the garbling scheme to satisfy *correctness* and *authenticity*. Since the circuit should only output 0 or 1 (depends on whether w is a valid witness) and verifier has no input, obliviousness and privacy are out of our consideration.

In addition to correctness and authenticity, we also require that the garbling scheme can be efficiently verified. That is, there exists an efficient algorithm Chk which takes as input a boolean circuit, a garbled circuit F and input wire label description E such that:

$$\mathsf{Chk}(f, F, E) \to D \text{ or } \perp$$

For our work, we require a variant garbling scheme different from the standard one,

- $\mathsf{Gb}(1^k, f, E, D) \to F$. Takes as input a boolean circuit f, descriptions of input wire labels E and output wire labels D, and outputs a garbled circuit F.
- $\mathsf{En}(E, x) \to X = E|_x$. Takes as input description of input wire labels E, a plaintext input x and outputs a garbled input X. In our schemes, encoding is always done via $E|_x$.
- $\mathsf{Ev}(F, X) \to Y$. Takes as input a garbled circuit F and a garbled input X and returns a garbled output Y.

[1] For each wire, E and D are wire labels includes both value 0 and 1, X and Y are wire labels only has value 0 or 1.

- $\mathsf{Chk}(f, F, E) \to D$ or $\perp$. Takes as input a boolean circuit, a (purported) garbled circuit F and input wire label description E and outputs either D or an error indicator $\perp$.

and also has a different security requirements:

Definition 8. *A garbling scheme satisfies* **correctness** *if:*

1. *For all circuits f, circuit-inputs x, and valid wire label descriptions E, D,*

$$\mathsf{Chk}(f, F, E) = D \text{ whenever } F \leftarrow \mathsf{Gb}(1^k, f, E, D)$$

2. *For all circuits f, (possibly malicious) garbled circuits F and wire-label descriptions E,*

$$\mathsf{Ev}(F, E|_x) = D|_{f(x)} \text{ whenever } \mathsf{Chk}(f, F, E) = D \neq \perp$$

Definition 9. *Let $\mathcal{W}$ denote the uniform distribution of $m \times 2$ matrices as described above. A garbling scheme has* **authenticity** *if for every circuit f, circuit-input x, and PPT algorithm $\mathcal{A}$, the following probability:*

$$\Pr[\exists y \neq f(x), \tilde{D} = D|_y : E \leftarrow \mathcal{W}, F \leftarrow \mathsf{Gb}(1^k, f, E, D), \tilde{D} = \mathcal{A}(F, E|_x)]$$

is negligible in k.

3.3 Protocol overview

Adapting to the ORAM setting, using constant rounds. We follow roughly the RAM-2PC paradigm of [19, 1], with some important differences. Let Π be an Oblivious RAM program with memory $\widehat{M}$, that implements $R(M, x, \cdot)$.[2] We assume a trusted setup phase in which Π's memory $\widehat{M}$ and state st are initialized from M. The prover learns $\widehat{M}$, st, as well as a garbled encoding of these values (i.e., one wire label for each bit of memory & state); the verifier specifies the garbled encoding to be used (i.e., both wire labels for each bit). If we follow [19, 1] strictly, we would have both parties repeatedly evaluate the next-memory-access circuit of Π, updating memory $\widehat{M}$, until it halts. However, this would result in a protocol with one round of interaction for each memory access of Π.

[2] We use M to refer to the logical RAM memory, and $\widehat{M}$ to refer to the physical ORAM memory.

To see how to achieve the same effect in a constant number of rounds, imagine that when executing an ORAM program, the memory access pattern $\mathcal{I}$ is known in advance. Then it is possible to express the entire computation in a single circuit. The circuit includes many copies of the RAM program's next-memory-access circuit, but is wired together under the assumption that the memory accesses will be $\mathcal{I}$. For example, if $\mathcal{I}$ says that Π writes to some memory block at time 2, and later reads from the same memory block at time 10, then the memory-output wires of subcircuit copy #2 will be connected to the memory-input wires of subcircuit copy #10, and so on.

We can leverage this optimization in our setting because the prover knows all (plaintext) inputs to Π, including the contents of memory and the ORAM state. Hence, the prover can execute Π locally to determine the complete memory access pattern $\mathcal{I}$. Since Π is an oblivious RAM, its access pattern $\mathcal{I}$ leaks no information about the inputs/memory/state, so the prover can safely send $\mathcal{I}$ to the verifier. Using $\mathcal{I}$, the verifier constructs a *single* garbled circuit $C_{x,\mathcal{I}}$ as described above. To prevent the prover from lying about the access pattern $\mathcal{I}$, the circuit recomputes the memory access pattern of Π and compares it to (hard-coded) $\mathcal{I}$.

Hence, this setting admits a constant-round solution based on ORAM, but avoiding tools like garbled RAM [38, 14] which incorporate expensive additional crypto circuitry into the garbled circuits.

Reusing M to perform many proofs. We follow the approach of [1], where the prover stores the ORAM memory and ORAM state encoded as wire labels from the various garbled circuits. The idea is that these wire labels can be reused directly as inputs to subsequent circuits, avoiding oblivious transfers for garbled circuit input. However, some modifications are required to adapt this idea to our setting.

After evaluating a garbled circuit, the prover holds a garbled output encoding of ORAM state & memory. The *authenticity* property of the garbling scheme guarantees that the prover knows at most one valid label per wire. As soon as the garbled circuit is opened, however, the prover learns both labels for each wire and authenticity is lost. The output wire labels are no longer useful for input to subsequent circuits, as the prover can now feed arbitrary garbled state/memory into subsequent garbled circuits. We need a mechanism to restore authenticity on all wire labels that may be later used (this includes the ORAM internal state as well as all memory locations that are read or written by the

garbled circuit).

Say the two wire labels on some output wire are y_0 and y_1, and that the prover knows only y_b. Let us call y_0 and y_1 the *temporary* wire labels, since they will soon be discarded. The verifier chooses a random function h from a strongly universal hash family. Just before the garbled circuit is opened (clobbering wire-label authenticity), the parties perform a *private function evaluation (PFE)*, where the prover gives y_b, the verifier gives h, and the prover learns $h(y_b)$. After the PFE, the garbled circuit can be opened, revealing y_0 and y_1.

Define $y'_0 = h(y_0)$ and $y'_1 = h(y_1)$ to be the *permanent* wire labels for this wire. At the time of the PFE, the prover could not have guessed y_{1-b}, and so learned the output of h on some point that was not y_{1-b}. From strong universality of h, even if y_{1-b} is later revealed, $y'_{1-b} = h(y_{1-b})$ is still random from the prover's point of view. Hence the PFE "transfers" the authenticity guarantee from the temporary wire labels y_0, y_1 to the permanent ones y'_0, y'_1, preserving authenticity even after both of y_0, y_1 are revealed. Hence, y'_0, y'_1 are safe to use as input wire labels to subsequent garbled circuits.

For technical reasons, the PFE needs to be committing with respect to the input h (so that the verifier can later "open" the h that was used). We suggest two efficient instantiations of committing-PFE for strongly universal families: one based on oblivious linear function evaluation (OLFE) [54] and one based on the string-select variant of OT presented in [28].

Note that all the PFE instances can be run in parallel hence, maintaining the constant round complexity of the overall protocol.

Eliminating the verifier's storage requirement. As described so far, the verifier is required to keep track of two wire labels for each bit of $\widehat{M}$, at all times. We can decrease this burden somewhat by letting the verifier derive these wire labels from a PRF. Let s be a seed to a PRF. For simplicity, suppose a wire label encoding truth value b on the jth bit of the ith memory block, last accessed at time t, is chosen as $\mathsf{PRF}(s, i\|j\|t\|b)$. In the actual protocol, the choice of wire labels is slightly more complicated.

Using this choice of wire labels, the verifier need only remember the last-access time of each block of $\widehat{M}$. However, this is still storage proportional to $|\widehat{M}|$. To reduce the storage even further, we "outsource" the maintenance of these last-access times to the prover. Let $T[i]$ denote the last-access time of block i. We let the prover store the array

T authenticated by a Merkle tree for which the verifier remembers only the root node.[3]

Whenever the verifier is about to garble a circuit, he must be reminded of $T[i]$ for each memory block i to be read by the RAM in its computation. We make the prover report each such $T[i]$ to the verifier, authenticating each value via the Merkle tree. The ORAM circuit performs some reads & writes in $\widehat{M}$, so T and the Merkle tree are updated accordingly, for each memory block that was accessed. Note that all accesses to the Merkle tree are done at the same time (in parallel), and similarly for the updates at the end of the execution.

Overall, accessing/updating the authenticated array T adds polylogarithmic (in $|\widehat{M}|$) communication/computation overhead and only a small constant number of rounds to the protocol. Instead of remembering two wire labels for each bit of $\widehat{M}$, the verifier need now remember only a PRF seed and the root of a Merkle tree.

3.4 Additional Notation and Helper Routines

ORAM components: Let $\mathcal{I}$ be an ORAM memory access sequence. We define $\mathsf{read}(\mathcal{I}) = \{i \mid (\text{READ}, i) \in \mathcal{I}\}$, $\mathsf{write}(\mathcal{I}) = \{i \mid (\text{WRITE}, i) \in \mathcal{I}\}$, and $\mathsf{access}(\mathcal{I}) = \mathsf{read}(\mathcal{I}) \cup \mathsf{write}(\mathcal{I})$; *i.e.*, the indices of blocks that are read/write/accessed in $\mathcal{I}$. If $S = \{s_1, \ldots, s_n\}$ is a set of memory-block indices, then we define $M[S] = (M[s_1], \ldots, M[s_n])$.

Let Π denote the next-instruction circuit of an ORAM. Given a zero-knowledge statement x and ORAM access sequence $\mathcal{I}$, we let circuit $C_{x,\mathcal{I}}$ denote the following circuit:

[3] More generally, T can be stored in any authenticated data structure that provides small storage for the verifier.

$\underline{C_{x,\mathcal{I}}(st, w, \widehat{M}[\mathsf{read}(\mathcal{I})]):}$
$(inst, st, block) := \Pi(st, (x, w), \bot)$
for $i = 1$ to $|\mathcal{I}| - 1$:
$r \leftarrow \{0,1\}^k$
if $\mathcal{I}[i] = (\text{READ}, id)$ then:
$(st, inst, \bot) \leftarrow \Pi(st, r, \widehat{M}[id])$
if $\mathcal{I}[i] = (\text{WRITE}, id)$ then:
$(st, inst, block) \leftarrow \Pi(\mathsf{st}, r, \bot)$
$\widehat{M}[id] = block$
$\mathcal{I}' := \mathcal{I}' \| inst$
$z := [\mathcal{I} \stackrel{?}{=} \mathcal{I}']$
return $(st, z, \widehat{M}[\mathsf{access}(\mathcal{I})])$

As described in last section, $C_{x,\mathcal{I}}$ is the circuit that will be garbled in the protocol. Note that both x and $\mathcal{I}$ are hard-coded into $C_{x,\mathcal{I}}$. Also, the circuit verifies that $\mathcal{I} = \mathcal{I}'$, and this entails checking the correctness of the witness since the final element of $\mathcal{I}$ is $(\text{HALT}, \text{TRUE})$.

Garbling notation: The circuit $C_{x,\mathcal{I}}$ has 3 logical inputs and 3 logical outputs, and we must distinguish among them. When garbling the circuit via $F \leftarrow \mathsf{Gb}(C_{x,\mathcal{I}}, E, D, 1^k)$, we denote by E a description of input wire labels (*i.e.*, two labels per wire) and D a description of output wire labels. We write $E = E_{\mathsf{st}} \| E_{\mathsf{wit}} \| E_{\mathsf{mem}}$, denoting the corresponding input wire labels for state, witness, and memory blocks, respectively. We define $D = D_{\mathsf{st}} \| D_{\mathsf{z}} \| D_{\mathsf{mem}}$ similarly. When referring to a specific memory block i, we use notation $E_{\mathsf{mem},i}$ and $D_{\mathsf{mem},i}$.

We use X to denote the prover's garbled input, and Y to denote the prover's garbled output (*i.e.*, one label per wire). As above, we define X_{st}, X_{wit}, X_{mem}, Y_{st}, Y_{z}, Y_{mem}. Finally, we have the prover maintain an array R_{mem} at all times, containing the current wire labels for all of the ORAM memory $\widehat{M}$.

For an overview of the notation used in the protocol, see Figure 3.3.

Temporary and permanent wire labels. Recall from last section that the output wire labels of a circuit are "temporary" in the sense that their authenticity is lost when

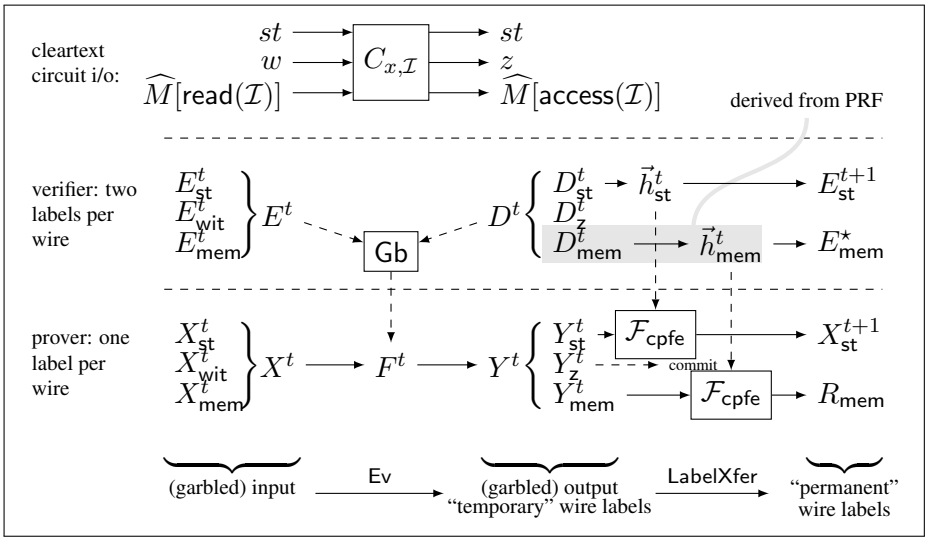

Figure 3.3: Summary of variables and notation used in the protocol.

the garbled circuit is opened. We use PFE to transfer the authenticity property of these temporary wire labels to a different set of "permanent" wire labels.

We transfer authenticity with the LabelXfer subprotocol, where Y is a list of "temporary" wire labels (*i.e.*, one label per wire), and $\vec{h}$ is a list of elements from a strongly universal hash family $\mathcal{H}$.

prot $\mathsf{LabelXfer}(Y, \vec{h})$:
  for $i = 1$ to $|Y|$ (in parallel):
    V sends $Y[i]$ and P sends $\vec{h}[i]$ to an instance of $\mathcal{F}_{\mathsf{cpfe}}$
    P receives output $Z[i] := \vec{h}[i](Y[i])$
  P outputs Z

Note that all instances of $\mathcal{F}_{\mathsf{cpfe}}$ are run in parallel and hence the protocol remains constant-round given that $\mathcal{F}_{\mathsf{cpfe}}$ is itself constant-round.

Selecting wire labels. Now let's consider how the verifier generates wire labels for the circuit. Recall from Section 3.3 that the verifier uses a PRF to generate wire labels corresponding to the ORAM memory, in order to reduce storage.

Since permanent wire labels are derived by applying strongly universal functions to

temporary wire labels, the verifier must also select strongly universal functions using the PRF to be able to reconstruct the choice of functions later.

Let s be the seed to a PRF. The verifier derives the *temporary* wire labels for a set S of memory block indices, last updated at time t, via the subroutine $\mathsf{TempMemLabels}$. The verifier derives the choice of strongly universal functions via the subroutine GenH.

Finally, the verifier derives the *current, permanent* wire labels for a set S of memory block indices via the subprotocol $\mathsf{PermMemLabels}$. Since each block may have been last accessed a different time, the authenticated array $\mathcal{F}_{\mathsf{Aut}}$ is referenced. For each block, the most recent temporary wire labels and strongly universal functions are reconstructed to derive the permanent wire labels.

func $\mathsf{TempMemLabels}(S, t)$:
  $D := \emptyset$
  for $i \in S$:
    for $j \in \{1, \ldots, l\}, b \in \{0, 1\}$:
      $D_i[j, b] = \mathsf{PRF}(s, 0 \| i \| j \| t \| b)$
    $D := D \| D_i$
  return D

func $\mathsf{GenH}(S, t)$:
  $\vec{h} = \emptyset$
  for $i \in S$:
    for $j \in \{1, \ldots, l\}$:
      $\vec{h}_i[j] = \mathsf{PRF}(s, 1 \| i \| j \| t)$
    $\vec{h} := \vec{h} \| \vec{h}_i$
  return $\vec{h}$

prot $\mathsf{PermMemLabels}(S)$:
  $E := \emptyset$
  for all i in S (in parallel):
    send $(\textsc{access}, i)$ to $\mathcal{F}_{\mathsf{Aut}}$
    receive $t_i := T[i]$
    $D_i := \mathsf{TempMemLabels}(\{i\}, t_i)$
    $\vec{h}_i := \mathsf{GenH}(\{i\}, t_i)$
    $E_i := \vec{h}_i(D_i)$
    $E := E \| E_i$
  return E

When $\vec{h}$ is an array of functions and D is a matrix of wire labels, the notation $\vec{h}(D)$ refers to the matrix E whose entries are $E[j, b] = \vec{h}[j](D[j, b])$.

3.5 Detailed protocol

Now we present the full protocol π. We refer to the prover as P and the verifier as V. The setup phase uses the initialization functionality $\mathcal{F}_{\mathsf{init}}$ defined in Figure 3.4.

- Initialize: On command (INIT, M) from P and $(\text{INIT}, D_{\mathsf{st}}, D_{\mathsf{mem}})$, where M is logical ORAM memory, and D_{st} & D_{mem} are wire label descriptions, run $(st, \widehat{M}) \leftarrow \mathsf{Initialize}(1^k, M)$. Give output $(st, \widehat{M}, D_{\mathsf{st}}|_{st}, D_{\mathsf{mem}}|_{\widehat{M}})$ to P.
- Open: On command OPEN from V, give output $(D_{\mathsf{st}}, D_{\mathsf{mem}})$ to P.

Figure 3.4: Ideal functionality $\mathcal{F}_{\mathsf{init}}$ for initializing an ORAM program along with wire labels.

Setup: On input M for prover P, let N denote the number of blocks in the ORAM encoding of M. Then both parties do the following:

1. V picks random wire label descriptions D^0_{st} and computes

$$D^0_{\mathsf{mem}} = \mathsf{TempMemLabels}([N], 0)$$

   V also chooses a random PRF seed $s \leftarrow \{0,1\}^k$.

2. P sends (INIT, M) to $\mathcal{F}_{\mathsf{init}}$; V sends $(\text{INIT}, D^0_{\mathsf{st}}, D^0_{\mathsf{mem}})$ to $\mathcal{F}_{\mathsf{init}}$. P receives output $(st, \widehat{M}, Y^0_{\mathsf{st}} = D^0_{\mathsf{st}}|_{st}, Y^0_{\mathsf{mem}} = D^0_{\mathsf{mem}}|_{\widehat{M}})$.

3. **[Transfer wire-label authenticity]:**[4]

   (a) V picks random vector $\vec{h}^0_{\mathsf{st}}$ of strongly universal functions and sets $E^1_{\mathsf{st}} = \vec{h}^0_{\mathsf{st}}(D^0_{\mathsf{st}})$. The parties perform subprotocol $\mathsf{LabelXfer}(Y^0_{\mathsf{st}}, \vec{h}^0_{\mathsf{st}})$, with P obtaining output $\vec{h}^0_{\mathsf{st}}(Y^0_{\mathsf{st}})$ which he stores as X^1_{st}.

   (b) V picks vector $\vec{h}^0_{\mathsf{mem}} = \mathsf{GenH}([N], 0)$ and the parties perform subprotocol $\mathsf{LabelXfer}(Y^0_{\mathsf{mem}}\vec{h}^0_{\mathsf{mem}})$. P receives output $\vec{h}^0_{\mathsf{mem}}(Y^0_{\mathsf{mem}})$ which he stores as R_{mem}.

   (c) V sends OPEN to $\mathcal{F}_{\mathsf{init}}$, and P receives output $(D^0_{\mathsf{st}}, D^0_{\mathsf{mem}})$.

[4] This step could be easily incorporated into $\mathcal{F}_{\mathsf{init}}$, but is written separately so that the remainder of the protocol has no edge-cases involving $t = 0$.

4. P sends (INIT, N) to $\mathcal{F}_{\mathsf{Aut}}$ to initialize authenticated array T (with $T[i] = 0$ for all i).

Proofs: On input (x, w) for the prover, let this be the tth such proof. The parties do the following:

4. **[ORAM Evaluation]:** P runs $\mathcal{I} \leftarrow \mathsf{RAMEval}(\Pi, \widehat{M}, x, w, st)$, then sends $(x, \mathcal{I})$ to V. V aborts if $\mathcal{I}$ is not an accepting access sequence. Note that $\mathsf{RAMEval}$ modifies $\widehat{M}$ for the prover.

5. **[Garbling the circuit]:** V generates a garbled circuit as follows:

   (a) V chooses input wire labels to the circuit as follows: E^t_{wit} are chosen randomly. E^t_{mem} are chosen as $E^t_{\mathsf{mem}} \leftarrow \mathsf{PermMemLabels}(\mathsf{read}(\mathcal{I}))$. Recall that E^t_{st} has been set previously.

   (b) V chooses output wire labels D^t_{z} and D^t_{st} randomly, and chooses $D^t_{\mathsf{mem}} = \mathsf{TempMemLabels}(\mathsf{access}(\mathcal{I}), t)$.

   (c) V sets $E^t = E^t_{\mathsf{st}} \| E^t_{\mathsf{wit}} \| E^t_{\mathsf{mem}}$, sets $D^t = D^t_{\mathsf{st}} \| D^t_{\mathsf{z}} \| D^t_{\mathsf{mem}}$, then invokes garbling algorithm $F^t \leftarrow \mathsf{Gb}(1^k, C_{x,\mathcal{I}}, E^t, D^t)$.

6. **[Evaluating garbled circuit]:**

   (a) The parties invoke $\mathcal{F}_{\mathsf{otc}}$ with P giving input w and V giving input E^t_{wit}. P receives $X^t_{\mathsf{wit}} = E^t_{\mathsf{wit}}|_w$. Additionally, P finds X^t_{st} in its memory and sets $X^t_{\mathsf{mem}} = R_{\mathsf{mem}}[\mathsf{read}(\mathcal{I})]$.

   (b) V sends F^t to P, and P evaluates the garbled circuit $Y^t \leftarrow \mathsf{Ev}(F^t, X^t)$.

   (c) P commits to Y^t_{z} (a single wire label) under $\mathcal{F}_{\mathsf{com}}$.

7. **[Transfer wire-label authenticity]:**

   (a) V picks random vector $\vec{h}^t_{\mathsf{st}}$ of strongly universal functions and sets $E^{t+1}_{\mathsf{st}} = \vec{h}^t_{\mathsf{st}}(D^t_{\mathsf{st}})$. The parties perform subprotocol $\mathsf{LabelXfer}(Y^t_{\mathsf{st}}, \vec{h}^t_{\mathsf{st}})$, with P obtaining output $\vec{h}^t_{\mathsf{st}}(Y^t_{\mathsf{st}})$ which he stores as X^{t+1}_{st}.

   (b) V picks vector $\vec{h}^t_{\mathsf{mem}} = \mathsf{GenH}(\mathsf{access}(\mathcal{I}), t)$ and the parties perform subprotocol $\mathsf{LabelXfer}(Y^t_{\mathsf{mem}}, \vec{h}^t_{\mathsf{mem}})$. P receives output $\vec{h}^t_{\mathsf{mem}}(Y^t_{\mathsf{mem}})$ which he stores as $R_{\mathsf{mem}}[\mathsf{access}(\mathcal{I})]$.

8. **[Check garbled circuit]:**
   (a) V sends OPEN to the $\mathcal{F}_{\mathsf{otc}}$-instance from time t, and P receives output E^t_{wit}.
   (b) V sends OPEN to the PFE-instances used for the state wire labels in time $t-1$. The prover thus obtains $\vec{h}^{t-1}_{\mathsf{st}}$ and sets $E^t_{\mathsf{st}} = \vec{h}^{t-1}_{\mathsf{st}}(D^{t-1}_{\mathsf{st}})$.
   (c) For each $i \in \mathsf{read}(\mathcal{I})$, verifier sends OPEN to the PFE-instances used for memory block i in time $T[i]$. The prover thus obtains $\vec{h}^{T[i]}_{\mathsf{mem},i}$ and sets $E^t_{\mathsf{mem},i} = \vec{h}^{T[i]}_{\mathsf{mem},i}(D^{T[i]}_{\mathsf{mem},i})$.
   (d) The verifier sets $E^t = E^t_{\mathsf{st}} \| E^t_{\mathsf{wit}} \| E^t_{\mathsf{mem}}$ and runs $D^t = \mathsf{Chk}(C_{x,\mathcal{I}}, F^t, E^t)$. If the result is $\perp$, then V aborts. Otherwise, V opens his commitment to Y^t_{z}.
9. **[Check prover's output]:** V checks whether $Y^t_{\mathsf{z}} = D^t_{\mathsf{z}}|_{\text{TRUE}}$. If not, then V aborts the protocol. Otherwise, V outputs (ACCEPT, t, x).
10. **[Update T]:** For all $i \in \mathsf{access}(\mathcal{I})$ (in parallel), V sends (UPDATE, i, t) to $\mathcal{F}_{\mathsf{Aut}}$.

Other discussion. Our protocol is written in a hybrid model with access to various setup functionalities. In particular, $\mathcal{F}_{\mathsf{cpfe}}$ is a *reactive* functionality, and our protocol involves many ($O(|\widehat{M}|)$) instances of $\mathcal{F}_{\mathsf{cpfe}}$ that remain "active" between ZK proofs. We have shown how the verifier's *inputs* to the $\mathcal{F}_{\mathsf{cpfe}}$ instances can be derived from a PRF, eliminating the need to explicitly store them. However, when these $\mathcal{F}_{\mathsf{cpfe}}$ instances are realized by concrete protocols, both parties are required to keep internal state between the PFE phase and opening phase. Hence, the verifier's *random coins* for the $\mathcal{F}_{\mathsf{cpfe}}$-protocols should also be derived from a PRF. In that way, the verifier's entire view can be reconstructed as needed when it is time to OPEN each $\mathcal{F}_{\mathsf{cpfe}}$ instance.

3.6 Security proof

Theorem 10. *The protocol π presented in Section 3.5 is a secure realization of the $\mathcal{F}^{\mathcal{R}}_{\mathsf{ZK}}$ functionality.*

Proof. We describe two simulators, depending on which party is corrupted.

Prover is corrupt: The primary role of the simulator in this case is to extract the witness from P. We construct the simulator in a sequence of hybrid interactions:

$\mathcal{H}_0$: Simulator plays the role of an honest verifier V (who has no input) and all ideal functionalities. In particular, the simulator obtains all of P's inputs to the ideal functionalities. This interaction is identical to the real interaction with π.

$\mathcal{H}_1$: Same as $\mathcal{H}_0$ except that instead of using a PRF, the simulated verifier chooses output wire labels D^t_{mem} and $\vec{h}^t_{\mathsf{mem}}$ functions uniformly (in TempMemLabels and GenH). We have $\mathcal{H}_1 \approx \mathcal{H}_0$ by the security of the PRF.

$\mathcal{H}_2$: Same as $\mathcal{H}_1$ except that the simulator aborts in certain cases as follows. The simulator has initially generated $\widehat{M}$ and st (while simulating $\mathcal{F}_{\mathsf{init}}$) and obtains w as P's input to $\mathcal{F}_{\mathsf{otc}}$ in each step (6a). Hence, each time in step 6, the simulator executes $C_{x,\mathcal{I}}(st, w, \widehat{M}[\mathsf{read}(\mathcal{I})]) \to (st, z, \widehat{M}[\mathsf{access}(\mathcal{I})])$, updating its internal st and $\widehat{M}$.

In the LabelXfer subprotocols in steps (3) and (7), P is meant to provide his garbled output Y^t_{mem} and Y^t_{st} to the $\mathcal{F}_{\mathsf{cpfe}}$ functionalities. Similarly, in step (6c), the prover is expected to commit to $Y^t_{\mathsf{z}}|_{\text{TRUE}}$. In $\mathcal{H}_2$, the simulator artificially aborts if P provides a *valid* encoding $D^t|_y$ for y not equal to the simulated output of $C_{x,\mathcal{I}}$ at time t.

Now we claim that the simulator artificially aborts with only negligible probability (so $\mathcal{H}_1 \approx \mathcal{H}_2$) and that the prover's view of E^t during step (7) in time t can be simulated given only $E^t_{\mathsf{mem}}|_{\widehat{M}[\mathsf{read}(\mathcal{I})]}$ and $E^t_{\mathsf{st}}|_{st}$. This follows essentially from the authenticity property of the garbling scheme and the strong-universal hashing property of $\mathcal{H}$.

Consider the LabelXfer subprotocol in step (3) (i.e., time $t = 0$). At this time, all wire labels in D^0 besides $D^0_{\mathsf{mem}}|_{\widehat{M}}$ and $D^0_{\mathsf{mem}}|_{st}$ are independent of the adversary's view by definition of the $\mathcal{F}_{\mathsf{init}}$ functionality. Hence, the simulator artificially aborts with negligible probability during these steps. Conditioned on not aborting, the action of the strongly universal hash functions on the "wrong" wire labels of D^0 — and hence the value of the "wrong" input wire labels in E^1 — is distributed independently of P's view. Thus P's view in step (6) can be simulated given only the claimed subset of E^1.

Inductively, the prover's view of E^t at the time of the LabelXfer steps depends only on the "expected" input wire labels. Hence, the simulator artificially aborts with

negligible probability, due to the authenticity property of the garbling scheme. As above, conditioned on not aborting, the strong universal hashing property ensures that the prover's view of E^{t+1} depends only on the claimed subset of E^{t+1}.

$\mathcal{H}_3$: Same as $\mathcal{H}_2$ except that in step (2) the simulator sends P's input M to $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$. In step (9), if the simulated verifier does not abort, then the simulator sends (x, w_{real}) to $\mathcal{F}_{\mathsf{ZK}}^{\mathcal{R}}$ (where w was extracted from the prover in step (6a). We claim that the output of the ideal verifier always matches that of the simulated verifier. The simulated verifier accepts the proof if P has committed to $D_{\mathsf{z}}^t|_{\text{TRUE}}$. Provided that the simulator has not artificially aborted, then it must be that the simulated $C_{x,\mathcal{I}}$ has output $z = \text{TRUE}$. By the correctness of the RAM program, it must be that w_{real} is a valid witness for x.

Hence, the simulator implicit in $\mathcal{H}_3$ is our final simulator.

Verifier is corrupt: In this case, the primary role of the simulator is to simulate its view without knowledge of the witness w. We note that the only information that needs to be simulated in each proof is the memory access sequence $\mathcal{I}$ and the opened commitment to output wire label Y_{z}^t. Again we proceed in a sequence of hybrid interactions.

$\mathcal{H}_0$: Simulator plays the role of an honest prover P (including M and witnesses w as input) and all ideal functionalities. Hence, the simulator obtains all of V's inputs to the ideal functionalities. This interaction is identical to the real interaction with π.

$\mathcal{H}_1$: Same as $\mathcal{H}_0$ except for the following changes. An honest prover computes D^t in step (8d) when the verifier decommits to certain inputs to ideal functionalities. Here we have the simulator perform the same computations, but as soon as possible given the ability to see the verifier's inputs to the functionalities. Hence, in step (6c), the simulator will know the entire contents of D^t. Instead of evaluating the garbled circuit to obtain garbled output Y_{z}^t, we have the simulator simply commit to $D_{\mathsf{z}}^t|_{\text{TRUE}}$.

This commitment is only opened when the garbled circuit F^t is shown to be correct. Hence, $\mathcal{H}_0 \equiv \mathcal{H}_1$.

$\mathcal{H}_2$: Same as $\mathcal{H}_1$ except for the following changes. Note that in $\mathcal{H}_1$ the simulator uses secret values M and w only to generate the memory access sequence $\mathcal{I}$. All of the simulated prover's other inputs to ideal functionalities can be set to dummy values, as V gets no outputs. So in $\mathcal{H}_2$ we have the simulated prover generate $\mathcal{I}$ in step (4) using the ORAM simulator instead of actually executing the RAM program itself. We have $\mathcal{H}_1 \approx \mathcal{H}_2$ by the security of the ORAM.

The simulator implicit in $\mathcal{H}_2$ defines our final simulator, since it no longer requires the secret values M and w to operate.

This completes the security proof of our protocol. □

Chapter 4: RAM Program with Malicious Security

4.1 Introduction

General secure two-party computation (2PC) allows two parties to perform "arbitrary" computation on their joint inputs without revealing any information about their private inputs beyond what is deducible from the output of computation. This is an extremely powerful paradigm that allows for applications to utilize sensitive data without jeopardizing its privacy.

From a feasibility perspective, we know that it is possible to securely compute any function, thanks to seminal results of [55, 16]. The last decade has also witnessed significant progress in design and implementation of more practical/scalable secure computation techniques, improving performance by orders of magnitude and enabling computation of circuits with billions of gates.

These techniques, however, are largely restricted to functions represented as Boolean or arithmetic circuits, whereas the majority of applications we encounter in practice are more efficiently captured using random-access memory (RAM) programs that allow constant-time memory lookup. Modern algorithms of practical interest (e.g., binary search, Dijkstra's shortest-paths algorithm, and the Gale-Shapely stable matching algorithm) all rely on fast memory access for efficiency, and suffer from major blowup in running time otherwise. More generally, a circuit computing a RAM program with running time T requires $\Theta(T^2)$ gates in the worst case, making it prohibitively expensive (as a general approach) to compile RAM programs into a circuit and then apply known circuit 2PC techniques.

A promising alternative approach uses the building block of *oblivious RAM*, introduced by Goldreich and Ostrovsky [18]. ORAM is an approach for making a RAM program's memory access pattern input-oblivious while still retaining fast (polylogarithmic) memory access time. Recent work in 2PC has begun to investigate direct computation of ORAM computations as an alternative to RAM-to-circuit compilation [19, 38, 26, 15, 37]. These works all follow the same general approach of evaluating a sequence of ORAM

instructions using traditional circuit-based 2PC phases. More precisely, they use existing circuit-based MPC to (1) initialize and setup the ORAM, a one-time computation with cost proportional to the memory size, (2) evaluate the next-instruction circuit which outputs "shares" of the RAM program's internal state, the next memory operations (read/write), the location to access, and the data value in case of a write. All of these existing solutions provide security only against semi-honest adversaries.

Challenges for malicious-secure RAM evaluation. It is possible to take a semi-honest secure protocol for RAM evaluation (e.g., [19]) and adapt it to the malicious setting using standard techniques. Doing so naïvely, however, would result in several major inefficiencies that are avoidable. We point out three significant challenges for efficient, malicious-secure RAM evaluation:

1: Integrity and consistency of state information, by which we mean both the RAM's small internal state and its large memory both of which are passed from one CPU step to the next. A natural approach for handling internal state is to have parties hold secret shares of the state (as in [19]), which they provide as input to a secure evaluation of the next-instruction circuit. Using standard techniques for malicious-secure SFE, it would incur significant overhead in the form of oblivious transfers and consistency checks to deal with state information as inputs to the circuit.

A natural approach suitable for handling RAM memory is to evaluate an Oblivious RAM that encrypts its memory contents. In this approach, the parties must evaluate a next-instruction circuit that includes both encryption and decryption sub-circuits. Evaluating a block cipher's circuit securely against malicious adversaries is already rather expensive [31], and this approach essentially asks the parties to do so at every time-step, even when the original RAM's behavior is non-cryptographic. Additional techniques are needed to detect any tampering of data by either participant, such as computing/verifying a MAC of each memory location access inside the circuit or computing a "shared" Merkle-tree on top of the memory in order to check its consistency after each access. All these solutions incur major overhead when state is passed or memory is accessed and are hence prohibitively expensive (see Appendix A for a concrete example).

2: Compatibility with batch execution and input-recovery techniques. In a secure computation, every input bit must be "touched" at some point. Oblivious RAM programs address this with a pre-processing phase that "touches" the entire (large) RAM

memory, after which the computation need not "touch" every bit of memory. Since an offline phase is already inevitable for ORAMs, we would like to use such a phase to further increase the efficiency of the online phase of the secure evaluation protocol. In particular, recent techniques of [21, 36] suggest that pre-processing/batching garbled circuits can lead to significant efficiency improvement for secure evaluation of circuits. The fact that the ORAM next-instruction circuits are used at every timestep and are known *a priori* makes the use of batch execution techniques even more critical.

Another recent technique, called input-recovery [33], reduces the number of garbled circuits in cut-and-choose by a factor of 3 by only requiring that at least one of the evaluated circuits is correct (as opposed to the majority). This is achieved by running an input-recovery step at the end of computation that recover's the garbler's private input in case he cheats in more than one evaluated circuit. The evaluator then uses the private input to do the computation on his own. A natural applications of this technique in case of RAM programs, would require running the input-recovering step after every timestep which would be highly inefficient (see Appendix A for a concrete example).

3: Run-time dependence. The above issues are common to any computation that involves persistent, secret internal state across several rounds of inputs/outputs (any so-called *reactive* functionality). RAM programs present an additional challenge, in that only part of memory is accessed at each step, and furthermore these memory locations are determined *only at run-time.* In particular, it is non-trivial to reconcile run-time data dependence with offline batching optimizations.

4.1.1 Our contribution

In a RAM computation, both the memory and internal state need to be *secret* and *resist tampering* by a malicious adversary. As mentioned above, the obvious solutions to these problem all incur major overhead whenever state is passed from one execution to the next or memory is accessed. We bypass all these overheads and obtain secrecy and tamper-resistance essentially for free. Our insight is that these are properties also shared by wire labels in most garbling schemes — they hide the associated logical value, and, given only one wire label, it is hard to "guess" the corresponding complementary label.

Hence, instead of secret-sharing the internal state of the RAM program between the

parties, we simply "re-use" the garbled wire labels from the output of one circuit into the input of the next circuit. These wire labels already inherit the required authenticity properties, so no oblivious transfers or consistency checks are needed.

Similarly, we also encode the RAM's memory via wire labels. When the RAM reads from memory location ℓ, we simply reuse the appropriate output wire labels from the most recent circuit to write to location ℓ (not necessarily the previous instruction, as is the case for the internal state). Since the wire labels already hide the underlying logical values, we only require an oblivious RAM that hides the memory access pattern and *not* the contents of memory. More concretely, this means that we do not need to add encryption/decryption and MAC/verify circuitry inside the circuit that is being garbled or perform oblivious transfers on shared intermediate secrets.

Importantly, if the RAM program being evaluated is "non-cryptographic" (i.e., has a small circuit description) then the circuits garbled at each round of our protocols will be small.

Of course, it is a delicate task to make these intuitive ideas work with the state of art techniques for cut-and-choose. We present two protocols, which use different approaches for reusing wire labels.

The first protocol uses ideas from the LEGO paradigm [42, 13] for 2PC and other recent works on batch-preprocessing of garbled circuits [21, 36]. The idea behind these techniques is to generate all the necessary garbled circuits in an offline phase (before inputs are selected), open and check a random subset, and randomly assign the rest into buckets, where each bucket corresponds to one execution of the circuit. But unlike the setting of [21, 36], where circuits are processed for many *independent* evaluations of a function, we have the additional requirement that the wire labels for memory and state data should be directly reused between various garbled circuits. Since we cannot know which circuits must have shared wire labels (due to random assignment to buckets and run-time memory access pattern), we use the "soldering" technique of [42, 13] that directly transfers garbled wire labels from one wire to another, after the circuits have been generated. However, we must adapt the soldering approach to make it amenable to soldering entire circuits as opposed to soldering simple gates as in [42, 13]. For a discussion of subtle problems that arise from a direct application of their soldering technique, see Section 4.3.

Our second approach directly reuses wire labels without soldering. As a result, gar-

bled circuits cannot be generated offline, but the scheme does not require the homomorphic commitments required for the LEGO soldering technique. At a high level, we must avoid having the cut-and-choose phase reveal secret wire labels that are shared in common with other garbled circuits. The technique recently proposed in [40] allows us to use a single cut-and-choose for all steps of the RAM computation (rather than independent cut-and-choose steps for each time step), and further hide the set of opened/evaluated circuits from the garbler using an OT-based cut-and-choose [25, 31]. We observe that this approach is compatible with the state of the art techniques for input-consistency check [39, 50].

We also show how to incorporate the input-recovery technique of [33] for reducing the number of circuits by a factor of three. The naive solution of running the cheating recovery after each timestep would be prohibitively expensive since it would require running a malicious 2PC for the cheating recovery circuit (and the corresponding input-consistency checks) at every timestep. We show a modified approach that only requires a final cheating recovery step at the end of the computation.

4.2 Preliminaries

4.2.1 Garbling Scheme

The garbling scheme we use in this work is slightly different from the standard garbling scheme as in section 2.3.3. Our 2PC protocol constructions re-use wire labels between different garbled circuits, so we define a specialized syntax for garbling schemes in which the input and output wire labels are pre-specified.

We represent a set of wire labels W as a $m \times 3$ array. Wire labels $W[i, 0]$ and $W[i, 1]$ denote the two wire labels associated with some wire i. We employ the point-permute optimization [43], so we require $\mathsf{lsb}(W[i, b]) = b$. The value $W[i, 2]$ is a single-bit *translation bit*, so that $W[i, W[i, 2]]$ is the wire label that encodes FALSE for wire i. For shorthand, we use $\tau(W)$ to denote the m-bit string $W[1, 2] \cdots W[m, 2]$.

We require the garbling scheme to have syntax $F \leftarrow \mathsf{Gb}(f, E, D)$ where f is a circuit, E and D represent wire labels as above.

For $v \in \{0, 1\}^m$, we define $W|_v = (W[1, v_1], \ldots, W[m, v_m])$, i.e., the wire labels with *select bits* v. We also define $W|_x^* := W|_{x \oplus \tau(W)}$, i.e., the wire labels corresponding to

truth values x. The correctness condition we require for garbling is that, for all f, x, and valid wire label descriptions E, D, we have:

$$\mathsf{Ev}(\mathsf{Gb}(F, E, D), E|_x^*) = D|_{f(x)}^*$$

If Y denotes a vector of output wire labels, then it can be decoded to a plain output via $\mathsf{lsb}(Y) \oplus \tau(D)$, where lsb is applied component-wise. Hence, $\tau(D)$ can be used as output-decoding information. More generally, if $\mu \in \{0,1\}^m$ is a mask value, then revealing $(\mu, \tau(D) \wedge \mu)$ allows the evaluator to learn only the output bits for which $\mu_i = 1$.

Let $\mathcal{W}$ denote the uniform distribution of $m \times 3$ matrices of the above form (wire labels with the constraint on least-significant bits described above). Then the security condition we need is that there exists an efficient simulator $\mathcal{S}$ such that for all f, x, D, the following distributions are indistinguishable:

$\underline{\text{Real}(f,x,D)}$:	$\underline{\text{Sim}^{\mathcal{S}}(f,x,D)}$:
$E \leftarrow \mathcal{W}$	$E \leftarrow \mathcal{W}$
$F \leftarrow \mathsf{Gb}(f, E, D)$	$F \leftarrow \mathcal{S}(f, E\|_x^*, D\|_{f(x)}^*)$
return $(F, E\|_x^*)$	return $(F, E\|_x^*)$

To understand this definition, consider an evaluator who receives garbled circuit F and wire labels $E|_x^*$ which encode its input x. The security definition ensures that the evaluator learns no more than the correct output wires $D|_{f(x)}^*$.

Consider what happens when we apply this definition with D chosen from $\mathcal{W}$ and against an adversary who is given only partial decoding information $(\mu, \tau(D) \wedge \mu)$.[1] Such an adversary's view is then independent of $f(x) \wedge \overline{\mu}$. This gives us a combination of the *privacy* and *obliviousness* properties of [6]. Furthermore, the adversary's view is independent of the complementary wire labels $D|_{\overline{f(x)}}^*$, except possibly in their least significant bits (by the point-permute constraint). So the other wire labels are hard to predict, and we achieve an *authenticity* property similar to that of [6].[2]

Finally, we require that it be possible to efficiently determine whether F is in the range of $\mathsf{Gb}(f, E, D)$, given (f, E, D). For efficiency improvements, one may also reveal

[1] Our definition applies to this case, since a distinguisher for the above two distributions is allowed to know D which parameterizes the distributions.

[2] We stress that the evaluator *can indeed decode* the garbled output (using $\tau(D)$ and the select bits), yet *cannot forge* valid output wire labels in their entirety. This combination of requirements was not considered in the definitions of [6].

a seed which was used to generate the randomness used in Gb.

These security definitions can be easily achieved using typical garbling schemes used in practice (e.g., [30]). We note that the above arguments hold even when the distribution $\mathcal{W}$ is slightly different. For instance, when using the Free-XOR optimization [30], wire label matrices E and D are chosen from a distribution parameterized by a secret Δ, where $E[i,0] \oplus E[i,1] = \Delta$ for all i. This distribution satisfies all the properties of $\mathcal{W}$ that were used above.

Conventions for wire labels. We exclusively garble the ORAM circuit which has its inputs/outputs partitioned into several logical values. When W is a description of input wire labels for such a circuit, we let $\mathsf{st}(W)$, $\mathsf{rand}(W)$, $\mathsf{block}(W)$ denote the submatrices of W corresponding to the incoming internal state, random tape, and incoming memory block. When W describes output wires, we use $\mathsf{st}(W)$, $\mathsf{inst}(W)$ and $\mathsf{block}(W)$ to denote the outgoing internal state, output instruction (read/write/halt, and memory location), and outgoing memory data block. We use these functions analogously for vectors (not matrices) of wire labels.

4.3 Batching Protocol

4.3.1 High-level Overview

Roughly speaking, the LEGO technique of [42, 13] is to generate a large quantity of garbled gates, perform a cut-and-choose on all gates to ensure their correctness, and finally assemble the gates together into a circuit which can tolerate a bounded number of faulty gates (since the cut-and-choose will not guarantee that all the gates are correct). More concretely, with sN gates and a cut-and-choose phase which opens half of them correctly, a statistical argument shows that permuting the remaining gates into **buckets** of size $O(s/\log N)$ each ensures that each bucket contains a majority of correct gates, except with negligible probability in s.

For each gate, the garbler provides a *homomorphic commitment* to its input/output wire labels, which is also checked in the cut and choose phase. This allows wires to be connected on the fly with a technique called **soldering.** A wire with labels (w_0, w_1) (here 0 and 1 refer to the public select bits) can be soldered to a wire with labels (w'_0, w'_1) as

follows. If w_0 and w'_0 both encode the same truth value, then decommit to $\Delta_0 = w_0 \oplus w'_0$ and $\Delta_1 = w_1 \oplus w'_1$. Otherwise decommit to $\Delta_0 = w_0 \oplus w'_1$ and $\Delta_1 = w_1 \oplus w'_0$. Then when an evaluator obtains the wire label w_b on the first wire, $w_b \oplus \Delta_b$ will be the correct wire label for the second wire. To prove that the garbler hasn't inverted the truth value of the wires by choosing the wrong case above, she must also decommit to the XOR of each wire's *translation* bit (i.e., $\beta \oplus \beta'$ where w_β and $w'_{\beta'}$ both encode false).

Next, an arbitrary gate within each bucket is chosen as the **head**. For each other gate, we solder its input wires to those of the head, and output wires to those of the head. Then an evaluator can transfer the input wire labels to each of the gates (by XORing with the appropriate solder value), evaluate the gates, and transfer the wire labels back. The majority value is taken to be the output wire label of the bucket. The cut-and-choose ensures that each bucket functions as a correct gate, with overwhelming probability. Then the circuit can be constructed by appropriately soldering together the buckets in a similar way.

For our protocol we use a similar approach but work with buckets of *circuits*, not buckets of gates. Each bucket evaluates a single timestep of the RAM program. To transfer RAM memory and internal state between timesteps, we solder wires together appropriately (i.e., state input of time t soldered to state output of time $t-1$; memory-block input t soldered to memory-block output of the previous timestep that wrote to the desired location). Additionally, the approach of using buckets also saves an asymptotic $\log T$ factor in the number of circuits needed for each timestep (i.e., the size of the buckets), where T is the total running time of the ORAM, a savings that motivates similar work on batch pre-processing of garbled circuits [21, 36].

We remark that our presentation of the LEGO approach above is a slight departure from the original papers [42, 13]. In those works, all gates were garbled using Free XOR optimization, where $w_0 \oplus w_1$ is a secret constant shared on all wires. Hence, we have only one "solder" value $w_0 \oplus w'_0 = w_1 \oplus w'_1$. If the sender commits to only the "false" wire label of each wire, then the sender is prevented from inverting the truth value while soldering ("false" is always mapped to "false"). However, to keep the offset $w_0 \oplus w_1$ secret, only one of the 4 possible input combinations of each gate can be opened in the cut-and-choose phase. The receiver has only a 1/4 probability of identifying a faulty gate. This approach does not scale to a cut-and-choose of entire circuits, where the number of possible input combinations is exponential. Hence our approach of forgoing

common wire offsets $w_0 \oplus w_1$ between circuits and instead committing to the translation bits. As a beneficial side effect, the concrete parameters for bucket sizes are improved since the receiver will detect faulty circuits with probability 1, not 1/4.

Back to our protocol, P_1 generates $O(sT/\log T)$ garblings of the ORAM's next-instruction circuit, and commits to the circuits and their wire labels. P_2 chooses a random half of these to be opened and aborts if any are found to be incorrect.

For each timestep t, P_2 picks a random subset of remaining garbled circuits and the parties assemble them into a bucket $\mathcal{B}_t$ (this is the $\mathsf{MkBucket}$ subprotocol) by having P_1 open appropriate XORs of wire labels, as described above. We can extend the garbled-circuit evaluation function Ev to $\mathsf{EvalBucket}$ using the same syntax. Then $\mathsf{EvalBucket}$ inherits the correctness property of Ev with overwhelming probability, for each of the buckets created in the protocol.

After a bucket is created, P_2 needs to obtain garbled inputs on which to evaluate it. See Figure 4.2 for an overview. Let X_t denote the vector of input wire labels to bucket $\mathcal{B}_t$. We use $\mathsf{block}(X_t), \mathsf{st}(X_t), \mathsf{rand}(X_t)$ to denote the sets of wire labels for the input memory block, internal state, and shares of random tape, respectively. The simplest wire labels to handle are the ones for internal state, as they always come from the previous timestep. We solder the output internal state wires of bucket $\mathcal{B}_{t-1}$ to the input internal state wires of bucket $\mathcal{B}_t$. Then if Y_{t-1} were the output wire labels for bucket $\mathcal{B}_{t-1}$ by P_2, we obtain $\mathsf{st}(X_t)$ by adjusting $\mathsf{st}(Y_{t-1})$ according to the solder values.

If the previous memory instruction was a READ of a location that was last written to at time t', then we need to solder the appropriate output wires from bucket $\mathcal{B}_{t'}$ to the corresponding input wires of $\mathcal{B}_t$. P_2 then obtains $\mathsf{block}(X_t)$ by adjusting the wire labels $\mathsf{block}(Y_{t'})$ according to the solder values. If the previous memory instruction was a READ of an uninitialized block, or a WRITE, then P_1 simply opens these input wire labels to all zero values (see $\mathsf{GetInput}_{\mathsf{pub}}$).

To obtain wire labels $\mathsf{rand}(X_t)$, we have P_1 open wire labels for its shares ($\mathsf{GetInput}_1$) and have P_2 obtain its wire labels via a standard OT ($\mathsf{GetInput}_2$).

At this point, P_2 can evaluate the bucket ($\mathsf{EvalBucket}$). Let Y_t denote the output wire labels. P_1 opens the commitment to their translation values, so P_2 can decode and learn these outputs of the circuit. P_2 sends these labels back to P_1, who verifies them for authenticity. Knowing only the translation values and not the entire actual output wire labels, P_2 cannot lie about the circuit's output except with negligible probability.

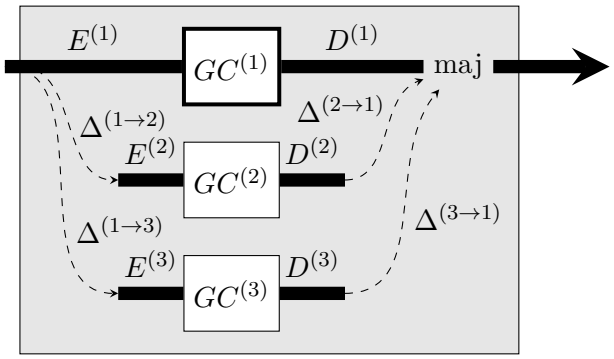

Figure 4.1: Illustration of $\mathsf{MkBucket}(\mathcal{B} = \{1, 2, 3\}, \mathsf{hd} = 1)$.

4.3.2 Detailed Protocol Description

Let Π be the ORAM program to be computed. Define $\tilde{\Pi}(\mathsf{st}, \mathsf{block}, \mathsf{inp}_1, \mathsf{inp}_{2,1}, \ldots, \mathsf{inp}_{2,n}) = \Pi(\mathsf{st}, \mathsf{block}, \mathsf{inp}_1, \bigoplus_i \mathsf{inp}_{2,i})$. Looking ahead, during the first timestep, the parties will provide $\mathsf{inp}_1 = x_1$ and $\mathsf{inp}_2 = x_2$, while in subsequent timesteps they input their shares r_1 and r_2 of the RAM program's randomness. P_2's input is further secret shared to prevent a selective failure attack on both x_2 and his random input r_2. We first define the following subroutines / subprotocols:

prot $\mathsf{Solder}(A, A')$ // A, A' are wire labels descriptions
  P_1 opens $\mathcal{F}_{\mathsf{xcom}}$-commitments to $\tau(A)$ and $\tau(A')$
    so that P_2 receives $\tau = \tau(A) \oplus \tau(A')$
  for each position i in τ and each $b \in \{0, 1\}$:
    P_1 opens $\mathcal{F}_{\mathsf{xcom}}$-commitments to $A[i, b]$ and $A'[i, \tau_i \oplus b]$
      so that P_2 receives $\Delta[i, b] = A[i, b] \oplus A'[i, \tau_i \oplus b]$
  return Δ

prot $\mathsf{MkBucket}(\mathcal{B}, \mathsf{hd})$ // $\mathcal{B}$ is a set of indices
  for each $j \in \mathcal{B} \setminus \{\mathsf{hd}\}$:
    $\Delta^{(\mathsf{hd}\to j)} = \mathsf{Solder}(E^{(\mathsf{hd})}, E^{(j)})$
    $\Delta^{(j\to\mathsf{hd})} = \mathsf{Solder}(D^{(j)}, D^{(\mathsf{hd})})$
  $\Delta^{(\mathsf{hd}\to\mathsf{hd})} :=$ all zeroes // for convenience

func $\mathsf{Adjust}(X, \Delta)$ // X is a vector of wire labels

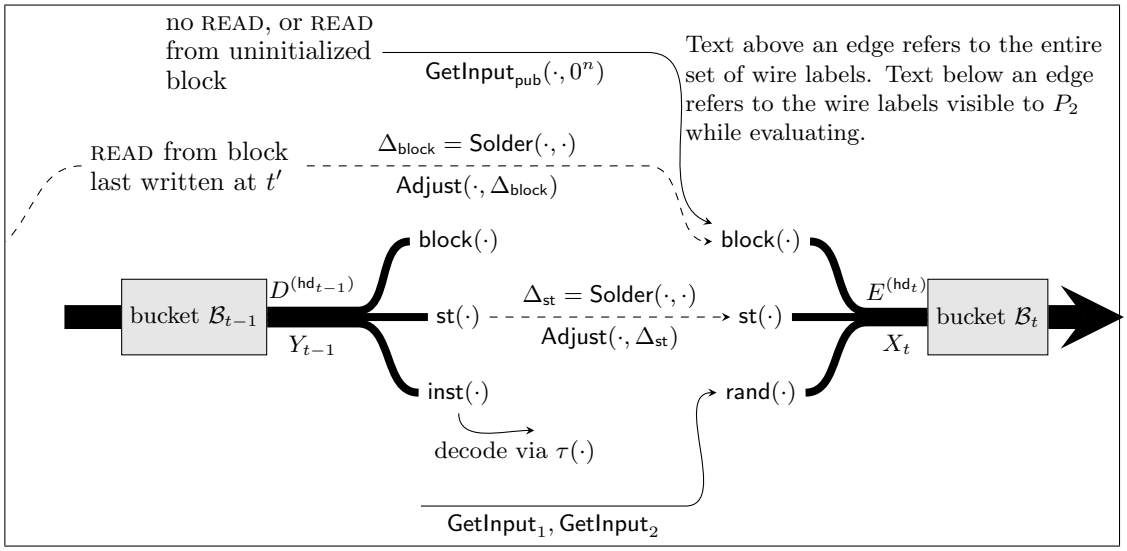

Figure 4.2: Overview of soldering and evaluation steps performed in the online phase.

for each i do $\tilde{X}[i] = X[i] \oplus \Delta[i, \mathsf{lsb}(X[i])]$
return $\tilde{X}$

func $\mathsf{EvalBucket}(\mathcal{B}, X, \mathsf{hd})$
for each j in $\mathcal{B}$:
$\tilde{X}_j = \mathsf{Adjust}(X, \Delta^{(\mathsf{hd} \to j))})$
$Y_j = \mathsf{Adjust}(\mathsf{Ev}(GC^{(j)}, \tilde{X}_j), \Delta^{(j \to \mathsf{hd})})$
return the majority element of $\{Y_j\}_{j \in \mathcal{B}}$

prot $\mathsf{GetInput}_{\mathsf{pub}}(A, x)$ // A describes wire labels; x public
P_1 opens commitments of $A|_x^*$; call the result X
P_1 opens commitments of $\tau(A)$
P_2 aborts if $\mathsf{lsb}(X) \neq \tau(A) \oplus x$; else returns X

prot $\mathsf{GetInput}_1(A, x)$ // A describes wire labels; P_1 holds x
P_1 opens commitments of $A|_x^*$; return these values

prot $\mathsf{GetInput}_2(A, x)$ // A describes wire labels; P_2 holds x
for each position i in A, parties invoke an instance of $\mathcal{F}_{\mathsf{ot}}$:

P_1 uses input $(A[i, A[i, 2]], A[i, 1 \oplus A[i, 2]])$
P_2 uses input x_i
P_2 stores the output as $X[i]$
P_2 returns X

We now describe the main protocol for secure evaluation of Π. We let s denote a statistical security parameter, and T denote an upper bound on the total running time of Π.

1. **[Pre-processing phase] Circuit garbling:** P_1 and P_2 agree on the total number $N = O(sT/\log T)$ of garbled circuits to be generated. Then, for each circuit index $i \in \{1, \ldots, N\}$:
   (a) P_1 chooses random input/output wire label descriptions $E^{(i)}$, $D^{(i)}$ and commits to each of these values component-wise under $\mathcal{F}_{\mathsf{xcom}}$.
   (b) P_1 computes $GC^{(i)} = \mathsf{Gb}(\tilde{\Pi}, E^{(i)}, D^{(i)})$ and commits to $GC^{(i)}$ under $\mathcal{F}_{\mathsf{com}}$.
2. **[Pre-processing phase] Cut and choose:** P_2 randomly picks a subset S_c of $\{1, \ldots, N\}$ of size $N/2$ and sends it to P_1. S_c will denote the set of check circuits and $S_e = \{1, \ldots, N\} \setminus S_c$ will denote the set of evaluation circuits. For check circuit index $i \in S_c$:
   (a) P_1 opens the commitments of $E^{(i)}$, $D^{(i)}$, and $GC^{(i)}$.
   (b) P_2 checks that $GC^{(i)} \in \mathsf{Gb}(\tilde{\Pi}, E^{(i)}, D^{(i)})$; if not, P_2 aborts.
3. **Online phase:** For each timestep t:
   (a) **Bucket creation:** P_2 chooses a random subset of $\mathcal{B}_t$ of S_e of size $\Theta(s/\log T)$ and a random head circuit $hd_t \in \mathcal{B}_t$. P_2 announces them to P_1. Both parties set $S_e := S_e \setminus \mathcal{B}_t$.
   (b) **Garbled input: randomness:** P_1 chooses random $r_1 \leftarrow \{0,1\}^n$, and P_2 chooses random $r_{2,1}, \ldots, r_{2,n} \leftarrow \{0,1\}^n$. P_2 sets

$$\mathsf{rand}_1(X_t) = \mathsf{GetInput}_1(\mathsf{rand}_1(E^{(\mathsf{hd}_t)}), r_1)$$
$$\mathsf{rand}_2(X_t) = \mathsf{GetInput}_2(\mathsf{rand}_2(E^{(\mathsf{hd}_t)}), r_{2,1} \cdots r_{2,n})$$

(c) **Garbled input: state:** If $t > 1$ then the parties execute:

$$\Delta_{\mathsf{st}} = \mathsf{Solder}(\mathsf{st}(D^{(\mathsf{hd}_{t-1})}), \mathsf{st}(E^{(\mathsf{hd}_t)}))$$

and P_2 sets $\mathsf{st}(X_t) := \mathsf{Adjust}(\mathsf{st}(Y_{t-1}), \Delta_{\mathsf{st}})$.

Otherwise, in the first timestep, let x_1 and x_2 denote the inputs of P_1 and P_2, respectively. For input wire labels W, let $\mathsf{st}_1(W), \mathsf{st}_2(W), \mathsf{st}_3(W)$ denote the groups of the internal state wires corresponding to the initial state $x_1 \| x_2 \| 0^n$. To prevent selective abort attacks, we must have P_2 encode his input as n-wise independent shares, as above. P_2 chooses random $r_{2,1}, \ldots, r_{2,n} \in \{0,1\}^n$ such that $\sum_i^n r_{2,i} = x_2$, and sets:[3]

$$\begin{aligned}\mathsf{st}(X_t) = {} & \mathsf{GetInput}_1(\mathsf{st}_1(E^{(\mathsf{hd}_t)}), x_1) \\ & \| \ \mathsf{GetInput}_2(\mathsf{st}_2(E^{(\mathsf{hd}_t)}), r_{2,1} \cdots r_{2,n}) \\ & \| \ \mathsf{GetInput}_{\mathsf{pub}}(\mathsf{st}_3(E^{(\mathsf{hd}_t)}), 0^n)\end{aligned}$$

(d) **Garbled input: memory block:** If the previous instruction $\mathsf{inst}_{t-1} = (\textsc{read}, \ell)$ and no previous $(\textsc{write}, \ell)$ instruction has happened, or if the previous instruction was not a READ, then the parties do

$$\mathsf{block}(X_t) = \mathsf{GetInput}_{\mathsf{pub}}(\mathsf{block}(E^{(\mathsf{hd}_t)}), 0^n)$$

Otherwise, if $\mathsf{inst}_{t-1} = (\textsc{read}, \ell)$ and t' is the largest time step with $\mathsf{inst}_{t'} = (\textsc{write}, \ell)$, then the parties execute:

$$\Delta_{\mathsf{block}} = \mathsf{Solder}(\mathsf{block}(D^{(\mathsf{hd}_{t'})}), \mathsf{block}(E^{(\mathsf{hd}_t)}))$$

Then P_2 sets $\mathsf{block}(X_t) := \mathsf{Adjust}(\mathsf{block}(Y_{t'}), \Delta_{\mathsf{block}})$.

(e) **Construct bucket:** P_1 and P_2 run subprotocol $\mathsf{MkBucket}(\mathcal{B}_t, \mathsf{hd}_t)$ to assem-

[3] We are slightly abusing notation here. More precisely, the parties are evaluating a slightly different circuit $\tilde{\Pi}$ in the first timestep than other timesteps. In the first timestep, it is P_2's input x_2 that is encoded randomly, whereas in the other steps it is P_2's share r_2 of the random tape. However, the difference between these circuits is only in the addition of new XOR gates, and only at the input level. When using the Free-XOR optimization, these gates can actually be added after the fact, so the difference is compatible with our pre-processing.

ble the circuits.

(f) **Circuit evaluation:** For each $i \in \mathcal{B}_t$, P_1 opens the commitment to $GC^{(i)}$ and to $\tau(\mathsf{inst}(D^{(i)}))$. P_2 does $Y_t = \mathsf{EvalBucket}(\mathcal{B}_t, X_t, \mathsf{hd}_t)$.

(g) **Output authenticity:** P_2 sends $\tilde{Y} = \mathsf{inst}(Y_t)$ to P_1. Both parties decode the output $\mathsf{inst}_t = \mathsf{lsb}(\tilde{Y}) \oplus \tau(\mathsf{inst}(D^{(\mathsf{hd}_t)}))$. P_1 aborts if the claimed wire labels $\tilde{Y}$ do not equal the expected wire labels $\mathsf{inst}(D^{(\mathsf{hd}_t)})|^*_{\mathsf{inst}_t}$. If $\mathsf{inst}_t = (\textsc{halt}, z)$, then both parties halt with output z.

4.3.3 Efficiency and Parameter Analysis

In the offline phase, the protocol is dominated by the generation of many garbled circuits, $O(sT/\log T)$ in all. In Appendix B we describe computation of the exact constant. As an example, for $T = 1$ million, and to achieve statistical security 2^{-40}, it is necessary to generate $10 \cdot T$ circuits in the offline phase.

In the online phase, the protocol is dominated by two factors: the homomorphic decommitments within the Solder subprotocol, and the oblivious transfers (in $\mathsf{GetInput}_2$) in which P_2 receives garbled inputs. For the former, we require one decommitment for each input and output wire label (to solder that wire to another wire) of the circuit $\tilde{\Pi}$. Hence the cost in each timestep is proportional to the input/output size of the circuit and the size of the buckets. Continuing our example from above ($T = 10^6$ and $s = 40$), buckets of size 5 are sufficient.

In Appendix B we additionally discuss parameter settings for when the parties open a different fraction (i.e., not 1/2) of circuits in the cut-and-choose phase. By opening a smaller fraction in the offline phase, we require fewer circuits overall, at the cost of slightly more circuits per timestep (i.e., slightly larger buckets) in the online phase.

We require one oblivious transfer per input bit of P_2 per timestep (independent of the size of buckets). P_2's input is split in an s-way secret share to assure input-dependent failure probabilities, leading to a total of sn OTs per timestep (where n is the number of random bits required by $\tilde{\Pi}$). However, online oblivious transfers are inexpensive (requiring only few symmetric-key operations) when instantiated via OT extension [22, 2], where the more expensive "seed OTs" will be done in the pre-processing phase. In Section 4.3.5 we suggest further ways to reduce the required number of OTs in the online phase.

Overall, the online overhead of this protocol (compared to the semi-honest setting) is dominated by the bucket size, which is likely at most 5 or 7 for most reasonable settings.

In terms of memory requirements, P_1 must store all pre-processed garbled circuits, and P_2 must store all of their commitments. For each bit of RAM memory, P_1 must store the two wire labels (and their decommitment info) corresponding to that bit, from the last write-time of that memory location. P_2 must store only a single wire label per memory bit.

4.3.4 Security Proof

In this section we prove the security of the batching protocol of Section 4.3.

Case 1: P_1 is corrupted. In this part, we are going to construct a simulator $\mathcal{S}$ progressively by using a standard hybrid argument. Let π_f denote the protocol of section 4.3.2. We begin by showing the real view of P_1 during the protocol and then constructing the simulator such that $\mathcal{S}$ can therefore simulate the whole protocol independent of P_2's input. We define $\mathcal{H}_0$ to be the real protocol π_f, i.e. P_1 and P_2 follow the protocol while $\mathcal{S}$ does not change anything, it acts the same as P_2. During the execution of π_f, the view of P_1 consists of

1. A random check circuits set S_c.
2. A random subset of $\mathcal{B}$ of S_e of size $\Theta(s/\log T)$.
3. The view in the standard oblivious transfer protocols when running protocol $\mathsf{GetInput}_2$. Also, notice that P_2 may abort during the execution of protocol $\mathsf{GetInput}_{\mathsf{pub}}$ and $\mathsf{GetInput}_2$, $\mathcal{S}$ needs to compute such abort probabilities which are independent of P_2's input.
4. At the end of π_f, P_1 receives a message $\tilde{Y} = \mathsf{inst}(Y_t)$.

We construct $\mathcal{S}$ that simulates all P_1's view of above. Since (a) and (b) does not depend on any of P_2's input, $\mathcal{S}$ can just behave the same as an honest P_2: For the cut-and-choose, $\mathcal{S}$ picks a random subset S_c and sends it to P_1, if any checking circuit in S_c fails, $\mathcal{S}$ abort the protocol. Also, at each timestep t, $\mathcal{S}$ chooses a random subset $\mathcal{B}$ and announces it

to P_1. Now we describe the simulation of the rest of P_1's view, via a sequence of hybrid interactions:

Hybrid $\mathcal{H}_0$: Ideal functionality: We define hybrid $\mathcal{H}_0$ to be the same as the real interaction, where the simulator $\mathcal{S}$ plays the role of an honest P_2 and also honestly plays the role of the ideal functionalities of $\mathcal{F}_{\mathsf{xcom}}$, $\mathcal{F}_{\mathsf{com}}$ and $\mathcal{F}_{\mathsf{ot}}$. One thing we highlight is that $\mathcal{S}$ can extract P_1's input and all wire labels from the ideal functionlities.

Hybrid $\mathcal{H}_1$: Ensure good buckets: At each timestep t, in step (3f) of **Circuit Evaluation**, $\mathcal{S}$ learns all garbled circuits and wire labels from the ideal functionality $\mathcal{F}_{\mathsf{com}}$ and $\mathcal{F}_{\mathsf{xcom}}$, even for evaluation circuits. So we define hybrid $\mathcal{H}_1$ to be identical to $\mathcal{H}_2$ except that $\mathcal{S}$ will abort if $\mathcal{B}_t$ does not have a majority of good circuits. Here, by "good" circuit we mean that its the circuit would be accepted by P_2 in checking phase if P_1 had opened it (along with its wire labels).

To show that $\mathcal{H}_1 \approx \mathcal{H}_0$, it suffices to show that the simulator aborts due to a bad bucket only with negligible probability.

In Appendix B, we define a value $\mathbb{B}^*(\rho, T, m)$, which is the probability that the adversary successfully generates m malicious circuits, P_2 does not abort in the cut-and-choose phase, and yet some $\mathcal{B}_t$ does not contain a majority of good circuits, when buckets have size ρ and there are T timesteps. This event corresponds exactly to the event that the simulator aborts in $\mathcal{H}_1$. We assume that ρ is chosen so that $\mathbb{B}^*(\rho, T, m) < 2^{-s}$, which is negligible.

Hybrid $\mathcal{H}_2$: Compute $\tilde{Y}$ differently: Define $\mathcal{H}_2$ to be the same as $\mathcal{H}_1$, except for the following changes. $\mathcal{S}$ extracts P_1's plain input x_1 from the ideal functionalities in the first timestep, then executes the RAM program Π on inputs (x_1, x_2) as $\mathrm{RamEval}(\Pi, M, x_1, x_2)$.

At each "Circuit evaluation" step of the protocol, where P_2 performs

$$Y_t = \mathsf{EvalBucket}(\mathcal{B}_t, X_t, \mathsf{hd}_t)$$

$\mathcal{S}$ instead computes $Y_t = D^{(\mathsf{hd}_t)}|^*_{(\mathsf{st},\mathsf{inst},\mathsf{block})}$, where $(\mathsf{st}, \mathsf{inst}, \mathsf{block})$ denote the internal variables defined in $\mathrm{RamEval}(\Pi, M, x_1, x_2)$ for the corresponding timestep.

Then we claim that $\mathcal{H}_2 \equiv \mathcal{H}_1$. This follows the correctness condition of garbling schemes. Specifically, the correctness condition for garbling schemes is:

$$\mathsf{Ev}(\mathsf{Gb}(F, E, D), E|_x^*) = D|_{f(x)}^*$$

Thus, if the majority circuits in bucket $\mathcal{B}_t$ are good (which is guaranteed in these hybrids), it is easy to see that the correctness condition extends to $\mathsf{EvalBucket}$as:

$$\mathsf{EvalBucket}(\mathcal{B}_t, E^{(\mathsf{hd}_t)}|_x^*, \mathsf{hd}_t) = D^{(\mathsf{hd}_t)}|_{f(x)}^*.$$

Then, one can verify that at each timestep t, the garbled inputs X_t to $\mathsf{EvalBucket}$ always encode the inputs to Π within RamEval, and the garbled outputs Y_t of $\mathsf{EvalBucket}$ always encode the outputs of Π within RamEval.

Hybrid $\mathcal{H}_3$: Selective abort: In subprotocol $\mathsf{GetInput}_2$, parties invoke an instance of a standard oblivious transfer protocol $\mathcal{F}_{\mathsf{ot}}$. However, P_1 can use malicious wire labels for oblivious transfer and cause P_2 to abort when execute protocol π_f. Then the probability of P_2 aborting depends on P_2's input.

Our protocol used the technique of [35] to deal with selective aborts: namely, we encoded P_2's input via s-way XOR shares. We define $\mathcal{H}_3$ to be identical to $\mathcal{H}_2$ except that $\mathcal{S}$ uses the technique of [35] to simulate the probability of P_2's aborts, by extracting P_1's inputs to $\mathcal{F}_{\mathsf{ot}}$. The analysis of [35] shows that $\mathcal{S}$ can simulate the probability of P_2's abort to within $\ell 2^{-s+1}$, where ℓ denotes the length of input and s is the security parameter. Hence $\mathcal{H}_3 \approx \mathcal{H}_2$.

Hybrid $\mathcal{H}_4$: Simulating ORAM memory accesses Let $\mathcal{S}_{ORAM}$ be the simulator from the security definition of ORAMs (Section 2.4).

Notice that $\mathcal{H}_3$ does not actually use all outputs of the RAM next-instruction circuit Π. In the output of $\text{RamEval}(\Pi, M, x_1, x_2)$, only $\mathcal{I}(\Pi, M, x_1, x_2)$ is used in $\mathcal{H}_3$, to generate $\tilde{Y}_t$ which is sent to P_1. Define $\mathcal{H}_4$ to be identical to $\mathcal{H}_3$ except that $\mathcal{S}$ uses the simulated access pattern of $\mathcal{S}_{ORAM}(1^\lambda, f(x_1, x_2))$. From the security of ORAM, we have that $\mathcal{H}_4 \approx \mathcal{H}_3$.

Now the simulator $\mathcal{S}$ described in hybrid $\mathcal{H}_4$ is a valid simulator in the ideal world. $\mathcal{S}$ does not require P_2's input x_2 — it only requires $f(x_1, x_2)$ which it can receive from

the ideal functionality.

Case 2: P_2 is corrupted: First we give a overview of P_2's real view in the protocol. Then we use a sequence of hybrids to construct $\mathcal{S}$ step by by step until eventually, $\mathcal{S}$ can implement the protocol independent of P_1's input. Consider the protocol, P_2's view consists of:

1. Commitments to all garbled circuits and wire labels under $\mathcal{F}_{\mathsf{com}}$ and $\mathcal{F}_{\mathsf{xcom}}$.
2. The set of check circuits with size ρT.
3. The set of evaluation circuits with size ρT.
4. At each timestep t, P_2 receives wire labels from $\mathsf{GetInput}_{\mathsf{pub}}$ and P_1's auxiliary input wire labels in subprotcols $\mathsf{GetInput}_1$.
5. At each timestep t, P_2 receives his auxiliary input wire labels from $\mathcal{F}_{\mathsf{ot}}$ before he can evaluate the bucket $\mathcal{B}_t$. Notice that at the end of the protocol, P_2 sends the output $\tilde{Y} = \mathsf{inst}(Y_t)$ to P_1. P_1 may abort if $\tilde{Y} \neq \mathsf{inst}(D^{(\mathsf{hd}[\mathcal{B}_t])})|^*_{\mathsf{inst}_t}$.

We now describe the sequence of hybrids: Let $\mathcal{H}_0$ be the real protocol π_f and we formally describe the simulator $\mathcal{S}$.

Hybrid $\mathcal{H}_0$: Ideal functionalities: We begin by letting $\mathcal{S}$ follow Π as an honest P_1 except that $\mathcal{S}$ also plays the role of all of the ideal functionalities.

Hybrid $\mathcal{H}_1$: Circuits: From P_2's view, we see that P_2 eventually receives a set of check circuits S_c and a set of evaluation circuits S_e, both of size ρT. In the real world, P_1 generates those garbled circuits and commits to all of them in step (1) of pre-processing phase. We define $\mathcal{H}_1$ to be the same as $\mathcal{H}_0$ except that, instead of letting $\mathcal{S}$ generate all circuits at the very beginning, we have $\mathcal{S}$ simulate the commitment messages in the pre-processing phase, but actually garble a circuit (honestly) only when its associated commitments are about to be opened opened.

It is not hard to see that $\mathcal{H}_1 \equiv \mathcal{H}_0$ since we only delay the time of constructing circuits and such construction is independent of P_1's input.

Hybrid $\mathcal{H}_2$: Visible wire labels: Now, we would like to generate simulated garbled circuits for the evaluation circuits, but before that we must know exactly which wire labels will be visible to P_2.

Recall that in hybrid $\mathcal{H}_1$, $\mathcal{S}$ chooses random translation bits $\tau(E)$ for the wire labels. Then in subprotocol $\mathsf{GetInput}_2$, P_2 specifies certain inputs v and receives $E|_v^* = E|_{\tau(E)\oplus v}$. Let $\lambda(E) = \tau(E) \oplus v$ denote these select bits which become "visible" to P_2.

We define $\mathcal{H}_2$ so that $\mathcal{S}$ first chooses $\lambda(E)$ at random. Then it arranges so that P_2 receives these wire labels from subprotocol $\mathsf{GetInput}_2$. At the same time, $\mathcal{S}$ still extracts P_2's input v and sets $\tau(E) = \lambda(E) \oplus v$ accordingly.

Similarly, in $\mathcal{H}_1$, P_2 chooses the translation bits $\tau(D)$ randomly for output wire labels D. Conversely, in $\mathcal{H}_2$, at the time that $\mathcal{S}$ actually garbles this circuit, $\mathcal{S}$ already knows what the logical input to this circuit will be. Hence, it can simulate the steps of RamEval and predict what the output v of this circuit will be. Hence it chooses $\lambda(D)$ at random and sets $\tau(D) = \lambda(D) \oplus v$ accordingly.

Also note that in subprotocol $\mathsf{Solder}(A, A')$, P_1 is supposed to open a commitment to $\tau(A)\oplus\tau(A')$. In this hybrid, however, we can replace $\tau(A)\oplus\tau(A') = \lambda(A)\oplus\lambda(A')$ since the protocol only solders wires that will carry the same logical value.

We have that $\mathcal{H}_1 \equiv \mathcal{H}_2$, since all the distributions involved are identical.

Hybrid $\mathcal{H}_3$: Simulated circuits: We define hybrid $\mathcal{H}_3$ to be the same as $\mathcal{H}_2$ except that $\mathcal{S}$ generates each evaluation circuit using the simulator $\mathcal{S}_{GC}$ from the security of garbling schemes. More concretely, for each evaluation circuit, instead of running $\mathsf{Gb}(\tilde{\Pi}, E, D)$, we run $\mathcal{S}_{GC}(\tilde{\Pi}, E|_{\lambda(E)}, D|_{\lambda(D)})$.

Then we have $\mathcal{H}_3 \approx \mathcal{H}_2$, by the security of the garbling scheme.

Hybrid $\mathcal{H}_4$: Simulated access pattern: Observe that in $\mathcal{H}_3$, the values $\lambda(A)$ are used to simulate the garbled circuits, but corresponding $\tau(A)$ values are no longer used in the Solder subprotocol. The only place $\tau(A)$ values are used is when P_1 reveals $\tau(\mathsf{inst}(D^{(\mathsf{hd}_t)}))$.

Hence, as $\mathcal{S}$ is simulating the steps of RamEval, the only values it actually uses in $\mathcal{H}_3$ are the access pattern $\mathcal{I}(\Pi, M, x_1, x_2)$. We define $\mathcal{H}_4$ to be identical, except

that $\mathcal{S}$ uses the simulated access pattern $\mathcal{S}_{ORAM}(1^\lambda, f(x_1, x_2))$. Then we have that $\mathcal{H}_4 \approx \mathcal{H}_3$ by the security of ORAM.

Finally, $\mathcal{H}_4$ describes a valid simulator $\mathcal{S}$ for the ideal model. It does not use P_1's input x_1 except to obtain $f(x_1, x_2)$ to provide as input to $\mathcal{S}_{ORAM}$.

4.3.5 Optimizations

Here we present a collection of further optimizations compatible with our 2PC protocols:

4.3.5.1 Hide only the input-dependent behavior

Systems like SCVM [37] use static program analysis to "factor out" as much input-independent program flow as possible from a RAM computation, leaving significantly less residual computation that requires protection from the 2PC mechanisms.

The backend protocol currently implemented by SCVM achieves security only against semi-honest adversaries. However, our protocols are also compatible with their RAM-level optimizations, which we discuss in more detail:

Special-purpose circuits. For notational simplicity, we have described our RAM programs via a *single* circuit Π that evaluates each timestep. Then Π must contain subcircuits for every low-level instruction (addition, multiplication, etc) that may ever be needed by this RAM program.

Instruction-trace obliviousness means that the choice of low-level instruction (e.g., addition, multiplication) performed at each time t does not depend on private input. The SCVM system can compile a RAM program into an instruction-trace-oblivious one (though one does not need full instruction-trace obliviousness to achieve an efficiency gain in 2PC protocols). For RAM programs with this property, we need only evaluate an (presumably much smaller) instruction-specific circuit Π_t at each timestep t.

For the batching protocol of Section 4.3, enough instruction-specific circuits must be generated in the pre-processing phase to ensure a majority of correct circuits in each bucket. However, we point out that buckets at different timesteps could certainly be different sizes! One particularly interesting use-case would involve a very aggressive pre-processing of the circuits involved in the ORAM construction (i.e., the logic translating

logical memory accesses to physical accesses), since these will dominate the computation and do not depend on the functionality being computed.[4] The bucket size / replication factor for these timesteps could be very low (say, 5), while the less-aggressively pre-processed instructions could have larger buckets. In this case, the plain-RAM internal state could be kept separate from the ORAM-specific internal state, and only fed into the appropriate circuits.

Along similar lines, we have for simplicity described RAM programs that require a random input tape at each timestep. This randomness leads to oblivious transfers within the protocol. However, if it is known to both parties that a particular instruction does not require randomness, then these OTs are not needed. For example, deterministic algorithms require randomness only for the ORAM mechanism. Concretely, tree-based ORAM constructions [51, 52, 9] require only a small amount of randomness and at input-indepenent steps.

Memory-trace obliviousness. Due to their general-purpose nature, ORAM constructions protect *all* memory accesses, even those that may already be input-independent (for example, sequantial iteration over an array). One key feature of SCVM is detecting which memory accesses are already input-independent and not applying ORAM to them. Of course, such optimizations to a RAM program would yield benefit to our protocols as well.

4.3.5.2 Reusing memory

We have described our protocols in terms of a single RAM computation on an initially empty memory. However, one of the "killer applications" of RAM computations is that, after an initial quasi-linear-time ORAM initialization of memory, future computations can use time sublinear in the total size of data (something that is impossible with circuits). This requires an ORAM-initialized memory to be reused repeatedly, as in [19].

Our protocols are compatible with reusing garbled memory. In particular, this can be viewed as a single RAM computation computing a reactive functionality (one that takes inputs and gives outputs repeatedly).

[4] Such pre-processing yields an instance of *commodity-based MPC* [3].

4.3.5.3 Other Protocol Optimizations

Storage requirements for RAM memory. In our cut-and-choose protocol, P_1 chooses random wire labels to encode bits of memory, and then has to remember these wire labels when garbling later circuits that read from those locations. As an optimization, P_1 could instead choose wire labels via $F_k(t, j, i, b)$, where F is a suitable PRF, t is the timestep in which the data was written, j is the index of a thread, i is the bit-offset within the data block, and b is the truth value. Since memory *locations* are computed at run-time, P_1 cannot include the memory location in the computation of these wire labels. Hence, P_1 will still need to remember, for each memory location ℓ, the last timestep t at which location ℓ was written.

Adaptive garbling. In the batching protocol, P_1 must commit to the garbled circuits and reveal them only after P_2 obtains the garbled inputs. This is due to a subtle issue of (non)adaptivity in standard security definitions of garbled circuits; see [4] for a detailed discussion. These commitments could be avoided by using an adaptively-secure garbling scheme.

Online/offline tradeoff. For simplicity we described our online/offline protocol in which P_1 generates many garbled circuits and P_2 opens exactly half of them. Lindell and Riva [36] also follow a similar approach of generating many circuits in an offline phase and assigning the remainder to random buckets; they also point out that changing the fraction of opened circuits results in different tradeoffs between the amount of circuits used in the online and offline phases. For example, checking 20% of circuits results in fewer circuits overall (i.e., fewer generated in the offline phase) but larger buckets (in our setting, more garbled circuits per timestep in the online phase).

4.4 Streaming Cut-and-choose Protocol

4.4.1 High-level Overview

The standard **cut-and-choose approach** is (for evaluating a *single circuit*) for the sender P_1 to garble $O(s)$ copies of the circuit, and receiver P_2 to request half of them to be opened. If all opened circuits are correct, then with overwhelming probability (in s)

a majority of the unopened circuits are correct as well.

When trying to apply this methodology to our setting, we face the challenge of feeding past outputs (internal state, memory blocks) into future circuits. Naïvely doing a separate cut-and-choose for each timestep of the RAM program leads to problems when reusing wire labels. Circuits that are opened and checked in time step t must have wire labels independent of past circuits (so that opening these circuits does not leak information about past garbled outputs). Circuits used for evaluation must be garbled with input wire labels *matching* output wire labels of past circuits. But the security of cut and choose demands that P_1 cannot know, at the time of garbling, which circuits will be checked or used for evaluation.

Our alternative is to use a technique suggested by [40] to perform a single cut-and-choose that applies to all timesteps. We make $O(s)$ independent **threads** of execution, where wire labels are directly reused only within a single thread. A cut-and-choose step at the beginning determines whether each *entire thread* is used for checking or evaluation. Importantly, this is done using an oblivious transfer (as in [25, 31]) so that P_1 does not learn the status of the threads.

More concretely, for each thread the parties run an oblivious transfer allowing P_2 to pick up either k_{check} or k_{eval}. Then at each timestep, P_1 sends the garbled circuit but also encrypts the *entire set* of wire labels under k_{check} and encrypts wire labels for only her input under k_{eval}. Hence, in check threads P_2 receives enough information to verify correct garbling of the circuits (including reuse of wire labels — see below), but learns nothing about P_1's inputs. In evaluation threads, P_2 receives only P_1's garbled input and the security property of garbled circuits applies. If P_1 behaves incorrectly in a *check thread*, P_2 aborts immediately. Hence, it is not hard to see that P_1 cannot cause a majority of evaluation threads to be faulty while avoiding detection in *all* check threads, except with negligible probability.

Reusing wire labels is fairly straight-forward since it occurs only within a single thread. The next circuit in the thread is simply garbled with input wire labels matching the appropriate output wire labels in the same thread (i.e., the state output of the previous circuit, and possibly the memory-block output wires of an earlier circuit). We point out that P_1 must know the previous memory instruction before garbling the next batch of circuits: if the instruction was $(\textsc{read}, \ell)$, then the next circuit must be garbled with wire labels matching those of the last circuit to write to memory location ℓ. Hence

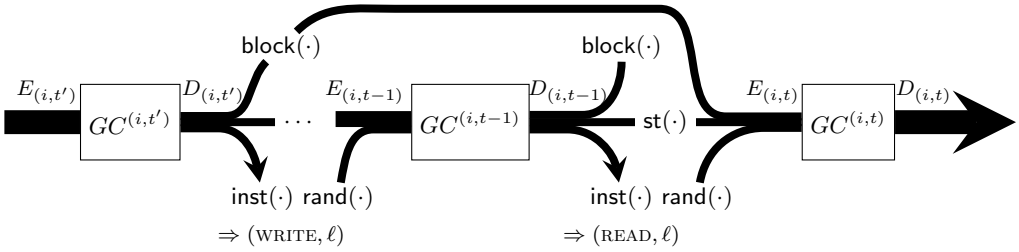

Figure 4.3: Wire-label reuse within a single thread i, in the streaming cut-and-choose protocol.

this approach is not compatible with batch pre-processing of garbled circuits.

For enforcing consistency of P_1's input, we use the approach of [50][5], where the very first circuit is augmented to compute a "hiding" universal hash of P_1's input. For efficiency purposes, the hash is chosen as $M \cdot (x_1 \| r)$, where M is a random binary matrix M of size $s \times (n + 2s + \log s)$ chosen by P_2. We prevent input-dependent abort based on P_2's input using the XOR-tree approach of [35], also used in the previous protocol.

We ensure authenticity of the output for P_1 using an approach suggested in [39]. Namely, wire labels corresponding to the same output wire and truth value are used to encrypt a random "output authenticity" key. Hence P_2 can compute these output keys only for the circuit's true output. P_2 is not given the information required for checking these ciphertexts until after he *commits* to the output keys. At the time of committing, he cannot guess complementary output keys, but he does not actually open the commitment until he receives the checking information and is satisfied with the check circuits.

The adaptation of the input-recovery technique of Lindell [33] is more involved and hence we discuss it separately in Section 4.4.5.

4.4.2 Detailed Protocol Description

We now describe the streaming cut-and-choose protocol for secure evaluation of Π, the ORAM program to be computed. Recall that $\tilde{\Pi}(\mathsf{st}, \mathsf{block}, \mathsf{inp}_1, \mathsf{inp}_{2,1}, \ldots, \mathsf{inp}_{2,n}) = \Pi(\mathsf{st}, \mathsf{block}, \mathsf{inp}_1 \bigoplus_i \mathsf{inp}_{2,i})$. We let s denote a statistical security parameter parameter, and T denote an upper bound on the total running time of Π. Here, we describe

[5] although our protocol is also compatible with the solution of [39].

the majority-evaluation variant of the protocol and discuss how to integrate the input-recovery technique in Section 4.4.5.

1. **Cut-and-choose.** The parties agree on $S = O(s)$, the number of threads (see discussion below). P_2 chooses a random string $b \leftarrow \{0,1\}^S$. Looking ahead, thread i will be a check thread if $b_i = 0$ and an evaluation thread if $b_i = 1$.

   For each $i \in \{1, \ldots, S\}$, P_1 chooses two symmetric encryption keys $k_{(i,check)}$ and $k_{(i,eval)}$. The parties invoke an instance of $\mathcal{F}_{\mathsf{ot}}$ with P_2 providing input b_i and P_1 providing input $(k_{(i,check)}, k_{(i,eval)})$.

2. **RAM evaluation.** For each timestep t, the following are done in parallel for each thread $i \in \{1, \ldots, S\}$:

   (a) **Wire label selection.** P_1 determines the input wire labels $E_{(t,i)}$ for garbled circuit $GC_{(t,i)}$ as follows. If $t = 1$, these wire labels are chosen uniformly. Otherwise, we set $\mathsf{st}(E_{(t,i)}) = \mathsf{st}(D_{(t-1,i)})$ and choose $\mathsf{rand}_1(E_{(t,i)})$ and $\mathsf{rand}_2(E_{(t,i)})$ uniformly. If the previous instruction $\mathsf{inst}_{t-1} = (\textsc{read}, \ell)$ and no previous $(\textsc{write}, \ell)$ instruction has happened, or if the previous instruction was not a $\textsc{read}$, then P_1 chooses $\mathsf{block}(E_{(t,i)})$ uniformly at random. Otherwise, we set $\mathsf{block}(E_{(t,i)}) = \mathsf{block}(D_{(t',i)})$, where t' is the last instruction that wrote to memory location ℓ.

   (b) **Input selection.** Parties choose shares of the randomness required for $\tilde{\Pi}$: P_1 chooses $r_1 \leftarrow \{0,1\}^n$, and P_2 chooses $r_{2,1}, \ldots, r_{2,n} \leftarrow \{0,1\}^n$.

   (c) **P_1's garbled input transfer.** P_1 sends the following wire labels, encrypted under $k_{(i,eval)}$:

   $$\begin{cases} \mathsf{st}_1(E_{(t,i)})|^*_{x_1} \text{ if } t = 1 \\ \mathsf{rand}_1(E_{(t,i)})|^*_{r_1} \end{cases}$$

   The following additional wire labels are also sent in the clear:

   $$\begin{cases} \mathsf{st}_3(E_{(t,i)})|^*_{0^n} \text{ if } t = 1 \\ \mathsf{block}(E_{(t,i)})|^*_{0^n} \text{ if } \textsc{write} \text{ or uninitialized } \textsc{read} \end{cases}$$

(d) **P_2's garbled input transfer.** P_2 obtains garbled inputs via calls to OT. To guarantee that P_2 uses the same input in all threads, we use a single OT across all threads for each input bit of P_2. For each input bit, P_1 provides the true and false wire labels for all threads as input to $\mathcal{F}_{\mathsf{ot}}$, and P_2 provides his input bit as the OT select bit.

Note that P_2's inputs consist of the strings $r_{2,1}, \ldots, r_{2,n}$ as well as the string x_2 for the case of $t = 1$.

(e) **Input consistency.** If $t = 1$, then P_2 sends a random $s \times (n + 2s + \log s)$ binary matrix M to P_1. P_1 chooses random input $r \in \{0,1\}^{2s+\log s}$, and augments the circuit for $\tilde{\Pi}$ with a subcircuit for computing $M \cdot (x_1 \| r)$.

(f) **Circuit garbling.** P_1 chooses output wire labels $D_{(t,i)}$ at random and does $GC^{(t,i)} = \mathsf{Gb}(\tilde{\Pi}, E_{(t,i)}, D_{(t,i)})$, where in the first timestep, $\tilde{\Pi}$ also contains the additional subcircuit described above. P_1 sends $GC^{(t,i)}$ to P_2 as well as $\tau(\mathsf{inst}(D_{(t,i)}))$.

In addition, P_1 chooses a random Δ_t for this time-step and for each inst-output bit j, he chooses random strings $w_{(t,j,0)}$ and $w_{(t,j,1)}$ (the same across all threads) to be used for output authenticity, such that $w_{(t,j,0)} \oplus w_{(t,j,1)} = \Delta_t$. For each thread i, output wire j and select bit b corresponding to truth value b', let $v_{i,j,b}$ denote the corresponding wire label. P_1 computes $c_{i,j,b} = \mathsf{Enc}_{v_{i,j,b}}(w_{(t,j,b')})$ and $h_{i,j,b} = H(c_{i,j,b})$, where H is a 2-Universal hash function. P_1 sends $h_{i,j,b}$ in the clear and sends $c_{i,j,b}$ encrypted under $k_{(eval,i)}$.

(g) **Garbled input collection.** If thread i is an evaluation thread, then P_2 assembles input wire labels $X_{(t,i)}$ for $GC^{(t,i)}$ as follows:

P_2 uses $k_{(eval,i)}$ to decrypt wire labels sent by P_1. Along with the wire labels sent in the clear and those obtained via OTs in $\mathsf{GetInput}_2$, these wire labels will comprise $\mathsf{rand}(X_{(t,i)})$; $\mathsf{block}(X_{(t,i)})$ in the case of a WRITE or uninitialized READ; and $\mathsf{st}(X_{(t,i)})$ when $t = 1$.

Other input wire labels are obtained via:

$$\mathsf{st}(X_{(t,i)}) = \mathsf{st}(Y_{(t-1,i)})$$
$$\mathsf{block}(X_{(t,i)}) = \mathsf{block}(Y_{(t',i)})$$

where t' is the last write time of the appropriate memory location, and Y denote the output wire labels that P_2 obtained during previous evaluations.

(h) **Evaluate and commit to output.** If thread i is an eval thread, then P_2 evaluates the circuit via $Y_{(t,i)} = \mathsf{Ev}(GC^{(t,i)}, X_{(t,i)})$ and decodes the output $\mathsf{inst}_{(t,i)} = \mathsf{lsb}(Y_{(t,i)}) \oplus \tau(D_{(t,i)})$. He sets $\mathsf{inst}_t = \text{majority}_i\{\mathsf{inst}_{(t,i)}\}$.

For each inst-output wire label j, P_2 decrypts the corresponding ciphertext $c_{i,j,b}$, then takes w'_j to be the majority result across all threads i. P_2 commits to w'_j.

If $t = 1$, then P_2 verifies that the output of the auxiliary function $M \cdot (x_1 \| r)$ is identical to that of all other threads; if not, he aborts.

(i) **Checking the check threads.** P_1 sends $\mathsf{Enc}_{k_{(i,check)}}(seed_{(t,i)})$ to P_2, where $seed_{(t,i)}$ is the randomness used in the call to Gb. Then if thread i is a check thread, P_2 checks the correctness of $GC^{(t,i)}$ as follows. By induction, P_2 knows all the previous wire labels in thread i, so can use $seed_{(t,i)}$ to verify that $GC^{(t,i)}$ is garbled using the correct outputs. In doing so, P_2 learns all of the output wire labels for $GC^{(t,i)}$ as well. P_2 checks that the wire labels sent by P_1 in the clear are as specified in the protocol, and that the $c_{i,j,b}$ ciphertexts and $h_{i,j,b}$ are correct and consistent. He also decrypts $c_{i,j,b}$ for $b \in \{0,1\}$ with the corresponding output label to recover $w'_{(t,j,b)}$ and checks that $w'_{(t,j,0)} \oplus w'_{(t,j,1)}$ is the same for all j. Finally, P_2 checks that the wire labels obtained via OT in $\mathsf{GetInput}_2$ are the correct wire labels encoding P_2's provided input. If any of these checks fail, then P_2 aborts immediately.

(j) **Output verification.** P_2 opens the commitments to values w'_j and P_1 uses them to decode the output inst_t. If a value w'_j does not match one of $w_{(t,j,0)}$ or $w_{(t,j,1)}$, then P_1 aborts.

4.4.3 Efficiency and Parameter Analysis

At each timestep, the protocol is dominated by the generation of S garbled circuits (where S is the number of threads) as well as the oblivious transfers for P_2's inputs. As before, using OT extension as well as the optimizations discussed in Section 4.3.5, the cost of the

oblivious transfers can be significantly minimized. Other costs in the protocol include simple commitments and symmetric encryptions, again proportional to the number of threads. Hence the major computational overhead is simply the number of threads. An important advantage of this protocol is that we avoid the soldering and the "expensive" xor-homomorphic commitments needed for input/outputs of each circuit in our batching solution. On the other hand, this protocol always require $O(s)$ garbled circuit executions regardless of the size of the RAM computation, while as discussed earlier, our batching protocol can require significantly less garbled circuit execution when the running time T is large. The choice of which protocol to use would then depend on the running time of the RAM computation, the input/output size of the next-instruction circuits as well as practical efficiency of xor-homomorphic commitment schemes in the future.

Compared to our other protocol, this one has a milder memory requirement. Garbled circuits are generated on the fly and can be discarded after they are used, with the exception of the wire labels that encode memory values. P_1 must remember $2S$ wire labels per bit of memory (although in Section 4.3.5 we discuss a way to significantly reduce this requirement). P_2 must remember between S and $2S$ wire labels per bit of memory (1 wire label for evaluation threads, 2 wire labels for check threads).

Using the standard techniques described above, we require $S \approx 3s$ threads to achieve statistical security of 2^{-s}. Recently, techniques have been developed [33] for the SFE setting that require only s circuits for security 2^{-s} (concretely, s is typically taken to be 40). We now discuss the feasibility of adapting these techniques to our protocol:

4.4.4 Security Proof

We assume an adversary $\mathcal{A}$ that can control any of the two parties (at most one party in a run of protocol). In what follows, we consider two cases: adversary controlling party P_1 or P_2.

1. P_1 **is corrupted.** Simulator $\mathcal{S}$ sets the simulated P_2's input as follows. It sets x_2 to all zeros since P_2's input can be anything. It will randomly choose the values for $r_{2,1}, \cdots, r_{2,n}$ as an honest P_2 would do, since the security of the ORAM depends on these values to be sampled randomly.

   Simulator would pick a random string b as an honest P_2 would and sets it as the

input of $\mathcal{F}_{\mathsf{ot}}$. The adversary will choose the two keys for each thread and sends them as his input to $\mathcal{F}_{\mathsf{ot}}$. Since $\mathcal{S}$ is simulating the $\mathcal{F}_{\mathsf{ot}}$, it will know both the "eval" and "check" keys for all the threads. Later on in the protocol, this will enable it to extract P_1's input.

At each time-step,

- $\mathcal{S}$ receives P_1's garbled input as described in the protocol. More specifically, for the first time-step $t = 1$, $\mathcal{S}$ receives $\mathsf{st}_1(E_{(t,i)})|^*_{x_1}$ and $\mathsf{rand}_1(E_{(t,i)})|^*_{r_1}$ encrypted under $k_{(i,eval)}$. Since the simulator already knows $k_{(i,eval)}$, it can decrypt them to extract the actual garbled value. To extract the actual input, simulator needs to know the opening of circuit. $\mathcal{S}$ will not know that until the check phase, which happens after the evaluation phase.
- $\mathcal{S}$ continues with the rest of the protocol as an honest P_2 would, choosing a random matrix M, gather the garbled input, evaluate the "eval" circuits, check the "check" circuits, and perform output verification. $\mathcal{S}$ will abort if an honest P_2 would have aborted.
- In checking phase, simulator will receive the seeds encrypted by $k_{(i,check)}$. Since it already knows $k_{(i,check)}$ for "all" the threads, it can extract P_1's input (for the first time-step, $t = 1$) as follows. $\mathcal{S}$ reconstruct the circuits of all "eval" threads using the seeds it had recovered. Afterwards, for the set of reconstructed eval circuits, it compares the input garbled values that it had received before against their corresponding circuits.

  If the garbled values match the opened circuits, $\mathcal{S}$ can extract P_1's input for that circuit. Simulator will then set P_1's input to be majority input to "eval" threads.

  Simulator will abort if either of the following events happen. 1) if the majority of "eval" circuits are bad (the reconstructed circuits are not valid garbling of the function that is being computed). 2) The majority of extracted inputs are invalid (if the garbled input values do not match the reconstructed circuits) or the valid input are inconsistent.

  Adversary can distinguish the simulator in the following cases. 1) The majority of the "eval" circuits are bad. In this case, an honest P_2 will not abort but

$\mathcal{S}$ will. Following the standard cut-and-choose arguments, this event happens with negligible probability. 2) All "eval" circuits are correct, the output of the hash function M is the same, but the inputs are inconsistent. In this case the honest P_2 will not abort but the simulator will. As discussed in [50], the probability of this event is negligible.

- Simulator will pass the extracted input of P_1 to TTP. It will then resume the protocol by performing the steps in checking phase and following the protocol for the rest of the time-steps, behaving as an honest P_2 would.
- To ensure that $\mathcal{A}$ cannot distinguish the block output of each time-step from a real execution, $\mathcal{S}$ create a sequence of simulated, random looking RAM accesses and in each time-step it returns one of them. Since the simulator has the seed to all the eval circuits of each time-steps, as describe above, it can return correct garbled values corresponding the simulated RAM access that it wishes to return. By security of ORAM, this simulated RAM access is indistinguishable from the actual execution.
- When the protocol finishes, $\mathcal{S}$ will then output whatever $\mathcal{A}$ outputs.

To prove the indistinguishability consider the following arguments.

- The simulator can abort in three cases: 1) if the output of the augmented circuits are not identical, or 2) if P_1 fails the checking phase. None of them depend on P_2's input. And 3) If inputs to "eval" threads are invalid, are inconsistent, or if the majority of "eval" circuits are bad circuits. As described above, in these cases $\mathcal{A}$ can distinguish the simulator but only with negligible probability.
- By security of ORAM, and the hiding property of the commitment scheme used, the choice of x_2 will not have a distinguishable effect on the view of $\mathcal{A}$ since all he sees during the run of the protocol are the commitments regarding the output authenticity and the memory access patterns. In particular, following the ORAM properties, memory access patterns look random in the view of the adversary and are indistinguishable regardless of P_2's input value.

2. P_2 **is corrupted.** Similar to the previous case, simulator sets x_2 to all zeros and assigns a random value to r_1. The rest of the simulation is as follows.

- $\mathcal{S}$ chooses random values for $k_{(i,eval)}$ and $k_{(i,check)}$ for all $i \in \{1, \ldots, S\}$ and sets them as input to $\mathcal{F}_{\mathsf{ot}}$. By simulating $\mathcal{F}_{\mathsf{ot}}$, $\mathcal{S}$ can extract P_1's choices of cut-and-choose bits.
- Simulator follows the protocol as an honest P_1 would do and selects garbled values for input wires, and sends the encrypted garbled values corresponding to his inputs as stated in the protocol.
- $\mathcal{S}$ will use the garbled values corresponding to P_2's input wires as input to $\mathcal{F}_{\mathsf{ot}}$. As before, since $\mathcal{S}$ is simulating $\mathcal{F}_{\mathsf{ot}}$, it will receive P_1's input when he passes them to $\mathcal{F}_{\mathsf{ot}}$. $\mathcal{S}$ will then pass P_1's input to TTP and receive the result of the computation z.
- In time-step $t = 1$, as instructed by the protocol, $\mathcal{S}$ will interact with P_2 to receive the matrix M. It would then choose r randomly.
- Having the matrix M, P_1's inputs, P_1's choices of cut-and-choose bits, and the result of computation z, $\mathcal{S}$ proceeds to garble the circuits as follows.
  (a) Simulator will create garbled circuits corresponding to checked threads as an honest P_1 would do. Simulator will also create the output authenticity values $w_{j,0}$ and $w_{j,1}$. And computes the values for $c_{i,j,b}$ and $h_{i,j,b}, b \in \{0, 1\}$ for "check" circuits as an honest P_1 would.
  (b) For the "eval" circuits, $\mathcal{S}$ behaves differently. In each time-step (except for the last), circuits should output some garbled value for st output wires (can be any arbitrary value) and a valid garbled value for block output wires. In the last time-step, the st output wires represent the output of the computation, so they cannot be arbitrary.
  $\mathcal{S}$ creates a series of random looking memory access instructions that it intents to output at each time-step. It also knows the values z of the last time-step st output wires. By security of garbling scheme, $\mathcal{S}$ can simulate garbled circuits that always output the garbled value corresponding the these predetermind values and leak nothing else.
- After garbling the circuits, $\mathcal{S}$ sends them along with output authenticity checks as stated above.
- It will continue the protocol to the end as an honest P_1 would and aborts accordingly.

The proof of indistinguishability is as follows.

- For input consistency check circuits, since P_1 is choosing the random values r and feeds $x_1||r$ to the hash function M, following [50] the output of the sub-circuit computing hash function M looks random.
- For the evaluation circuits, by security of the garbling scheme, $\mathcal{A}$ can guess the actual values of the garbled st values, with negligible probability. By security of the garbling scheme, if $\mathcal{A}$ knows one of the two garbled values of wire, he can correctly guess the other value only with negligible probability. Therefore, even though $\mathcal{A}$ will know the truth value of the garbled value corresponding to block output wires, he cannot obtain the other garbled value. Therefore, by security of the encryptions used, he cannot decrypt the $c_{i,j,1-b}$ since he does have access to the decryption key. As a result, $\mathcal{A}$ cannot distinguish the fake circuit from the correct circuit, except with negligible probability.

  For the last time-step, we can employ the same reasoning about the indistiguishability of the fake circuit that always outputs z with the actual circuit that computes z.
- Moreover, by security of the ORAM, the randomly created access patterns are indistinguishable from the real run of the protocol.
- The check circuits are constructed correctly and by security of Yao's protocol they do not leak any information regarding P_1's input. Therefore, they do not effect the view of the $\mathcal{A}$.
- In the rest of the simulation $\mathcal{S}$ acts as an honest P_1 would and aborts accordingly.

4.4.5 Integrating Cheating Recovery

The idea of [33] is to provide a mechanism that would detect inconsistency in the output wire labels encoding the final output of the computation. If P_2 receives output wire labels for two threads encoding disparate values, then a secondary computation allows him to recover P_1's input (and hence compute the function himself). This technique reduces the number of circuits necessary by a factor of 3 since we only need a single honest thread among the set of evaluated threads (as opposed to a majority). We refer

the reader to [33] for more details. We point out that in some settings, recovering P_1's input may not be enough. Rather, if P_2 is to perform the entire computation on his own in the case of a cheating P_1, then he also needs to know the contents of the RAM memory!

Cheating recovery at each timestep. It is possible to adapt this approach to our setting, by performing an input-recovery computation at the end of each timestep. But this would be very costly, since each input-recovery computation is a maliciously secure 2PC that requires expensive input-consistency checks for both party's inputs, something we worked hard to avoid for the state/memory bits. Furthermore, each cheating-recovery garbled circuit contains non-XOR gates that need to be garbled/evaluated $3s$ times at each timestep. These additional costs can become a bottleneck in the computation specially when the next-instruction circuit is small.

Cheating recovery at the end. It is natural to consider delaying the input-recovery computation until the last timestep, and only perform it once. If two of the threads in the final timestep (which also computes the final output of computation) output different values, the evaluator recovers the garbler's input. Unfortunately, however, this approach is not secure. In particular, a malicious P_1 can cheat in an intermediate timestep by garbling one or more incorrect circuits. This could either lead to two or more valid memory instruction/location outputs, or no valid outputs at all. It could also lead to a premature "halt" instruction. In either case, P_2 cannot yet abort since that would leak extra information about his private input. He also cannot continue with the computation because he needs to provide P_1 with the next instruction along with proof of its authenticity (i.e. the corresponding garbled labels) but that would reveal information about his input.

We now describe a solution that avoids the difficulties mentioned above and at the same time eliminates the need for input-consistency checks or garbling/evaluating non-XOR gates at each timestep. In particular, we delay the "proof of authenticity" by P_2 for all the memory instructions until after the last timestep. Whenever P_2 detects cheating by P_1 (i.e. more than two valid memory instructions), instead of aborting, he pretends that the computation is going as planned and sends "dummy memory operations" to P_1 but does not (and cannot) prove the authenticity of the corresponding wire labels yet.

For modern tree-based ORAM constructions ([52, 9], etc) the memory access pattern is always uniform, so it is easy for P_2 to switch from reporting the real memory access pattern to a simulated one. Note that in step (h) of the protocol, P_2 no longer needs to commit to the majority w'_j. As a result, step (j) of the protocol will be obsolete. Instead, in step (h), P_2 sends the inst_t in plaintext. This instruction is the single valid instruction he has recovered or a dummy instruction (if P_2 has attempted to cheat).

After the evaluation of the final timestep, we perform a fully secure 2PC for an input-recovery circuit that has two main components. The first one checks if P_1 has cheated. If he has, it reveals P_1's input to P_2. The second one checks the proofs of authenticity of the inst instructions P_2 reveals in all timesteps and signals to P_1 to abort if the proof fails.

First cheating recovery, then opening the check circuits. For this cheating recovery method to work, we perform the evaluation steps (step (h)) for all time-steps first (at this stage, P_2 only learns the labels for the final output but not the actual value), then perform the cheating recovery as described above, and finally perform all the checks (step (i)) for all time-steps.

We now describe the cheating recovery circuit which consists of two main components in more detail.

- The first component is similar to the original cheating recovery circuit of [33]. P_2's input is the XOR of two valid output authenticity labels for a wire j at step t for which he has detected cheating (if there is more than one instance of cheating he can use the first occurrence). Lets denote the output authenticity labels for jth bit of $\mathsf{block}(Y_{(t,i)})$ at time-step t with $w_{(t,j,b)}, b \in \{0,1\}$. Then P_2 will input $w_{(t,j,0)} \oplus w_{(t,j,1)}$ to the circuit. If there is no cheating, he inputs garbage. Notice that $w_{(t,j,0)} \oplus w_{(t,j,1)} = \Delta_t$ for valid output authenticity values, as described in the protocol (note that we assume that all output authenticity labels in timestep t use the same offset Δ_t).

  P_1 inputs his input x_1. He also hardcodes Δ_t. For timestep t (as shown in Figure 4.4) the circuit compares P_2's input against the hardcoded Δ_t. If P_2's input is the same as the Δ_t, cheating is detected and the circuit outputs 1. To check that P_2's input is the same as at least one of the hard-coded Δs, in the circuit of Figure 4.5

we compute the OR of all these outputs. Thus, if the output of this circuit is 1, it means that P_1 has cheated in at least one timestep.

To reveal P_1's input, we compute the AND of output of circuit of Figure 4.5 with each bit of P_1's input as depicted in Figure 4.6. This concludes the description of the first component for cheating recovery.

- In the second component, we check the authenticity of the memory instructions P_2 provided in all timesteps. In particular, he provides the hash of concatenation of all output authentication labels he obtained during the evaluation corresponding to inst in all timesteps (P_2 uses dummy labels if he does not have valid ones due to P_1's cheating), while P_1 does the same based on the plaintext instructions he received from P_2 and the labels which he knows. The circuit then outputs 1 if the two hash values match. The circuit structure is therefore identical to that of Figure 4.4, but the inputs are the hash values. An output of 0 would mean that P_2 does not have a valid proof of authenticity.

  As shown in the final circuit of Figure 4.6 then, if P_1 was not already caught cheating in the previous step, and P_2's proof of authenticity fails, the circuit outputs a 1 to signal an abort to P_1. This is a crucial condition, i.e., it is important to ensure P_1 did not cheat (the output of circuit of Figure 4.5) before accusing P_2 of cheating, since in case of cheating by P_1 say in timestep t, P_2 may be able to prove authenticity of the instructions for timestep t or later.

Efficiency: Following the techniques of [33], all the gates of Figures 4.4, and 4.5 can be garbled using non-cryptographic operations (XORs) and only the circuit of Figure 4.6 has non-XOR gates. More precisely it requires $|x_1|$ ANDs and a NOT gate.

Of course, the final circuit will be evaluate using a basic maliciously secure 2PC. Thus, we need to add a factor of $3s$ to the above numbers which results in garbling a total of $3s(|x_1|+1)$ non-XOR gates which is at most $12s(|x_1|+1)$ symmetric operations.

The input consistency checks are also done for P_1's input x_1 and P_2's input which is a proof of cheating of length $|\Delta|$ and a proof of authenticity which is the output of a hash function (both are in the order of the computational security parameter). We stress that the gain is significant since both the malicious 2PC and the input consistency cheks are only done once at the end.

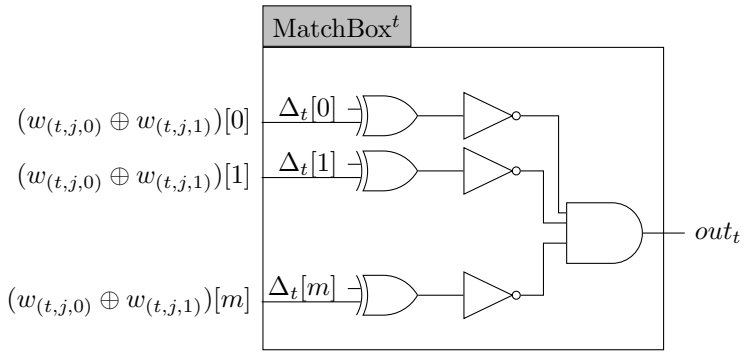

Figure 4.4: Cheating recovery component 1: MatchBox. Where $\Delta_t[i]$ denotes the ith bit of Δ_t and $m = |\Delta_t|$.

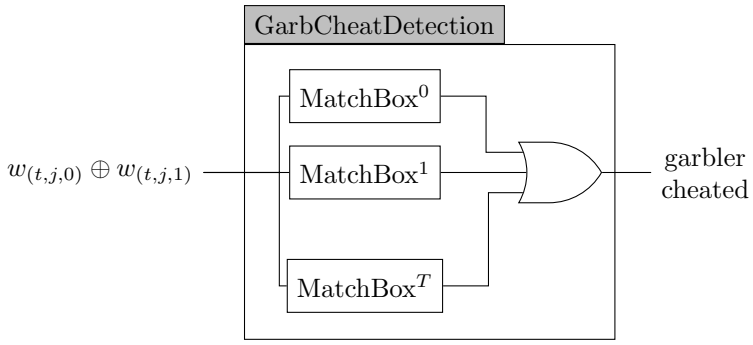

Figure 4.5: Cheating Recovery component 1: Garbler Cheating Detection.

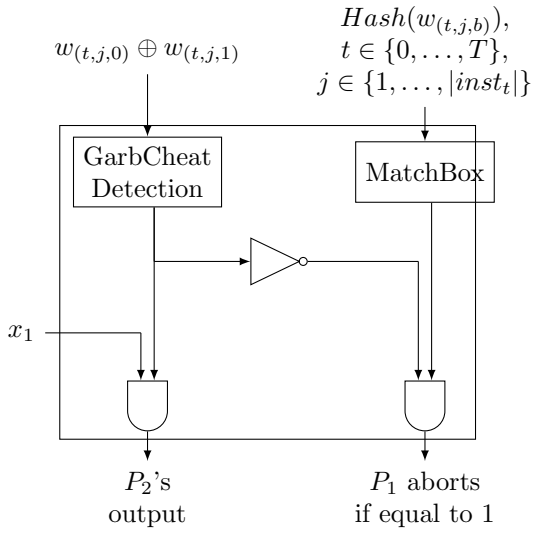

Figure 4.6: Final Circuit

Chapter 5: Conclusion

In this thesis, we have studied the role of oblivious RAM in secure multi-party computation. As we have mentioned above, most techniques are restricted to functions represented as boolean or arithmetic circuits, and the the conversion from function to circuit may lead to a huge blowup both in circuit size and running time. In the real world, modern algorithms of practical interest all rely on fast memory access for efficiency, and suffer from major blowup in running time otherwise. For example, consider performing a binary search on a dataset of size n, we only need to touch $O(\log n)$ data when applying RAM programs. However, when using garbled circuit technique, we have to touch all data which significantly increases the circuit size and running time. More generally, a circuit computing a RAM program with running time T requires $\Theta(T^2)$ gates in the worst case, making it prohibitively expensive (as a general approach) to compile RAM programs into a circuit and then apply known circuit 2PC techniques.

First we describe a system in which a prover holds a large dataset M and can repeatedly prove NP relations about that dataset. It is not appropriate here to use garbled circuit due to the large dataset. ORAM technique can help reducing the cost to sublinear in an amortized way. It requires only a constant number of rounds of interaction, incurs online computation and communication cost that is linear in the running time of the RAM program.

We then present the first practical protocols for evaluating RAM programs with security against malicious adversaries. We point out three main challenges in malicious-secure RAM evaluation, integrity and consistency of state information, compatibility with batch execution and input-recovery techniques, and run-time dependence. Then we introduce two protocols that address the issues. The first protocol uses the LEGO paradigm of label soldering for 2 party computation. The second approach directly reuses wire labels without soldering while garbled circuits must be generated online.

Based on some concrete measurements in Appendix A (see table A.1), the "extra overhead" of achieving malicious security for RAM programs (i.e. the additional cost beyond what is needed for malicious security of the circuits involved in the computation),

is at least an order of magnitude smaller than the naive solutions and this gap grows as the running time of the RAM program increases.

Bibliography

[1] Arash Afshar, Zhangxiang Hu, Payman Mohassel, and Mike Rosulek. How to efficiently evaluate RAM programs with malicious security. In *Eurocrypt*, 2015. To appear.

[2] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 535–548. ACM Press, November 2013.

[3] Donald Beaver. Commodity-based cryptography (extended abstract). In *29th Annual ACM Symposium on Theory of Computing*, pages 446–455. ACM Press, May 1997.

[4] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 134–153. Springer, December 2012.

[5] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. Cryptology ePrint Archive, Report 2012/265, 2012. http://eprint.iacr.org/2012/265.

[6] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12: 19th Conference on Computer and Communications Security*, pages 784–796. ACM Press, October 2012.

[7] Jan Camenisch and Markus Michels. Proving in zero-knowledge that a number is the product of two safe primes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 107–122. Springer, May 1999.

[8] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[9] Kai-Min Chung and Rafael Pass. A simple ORAM. Cryptology ePrint Archive, Report 2013/243, 2013. http://eprint.iacr.org/2013/243.

[10] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, pages 73–80, New York, NY, USA, 1972. ACM.

[11] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo Desmedt, editor, *Advances in Cryptology – CRYPTO'94*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer, August 1994.

[12] Cynthia Dwork, Uriel Feige, Joe Kilian, Moni Naor, and Shmuel Safra. Low communication 2-prover zero-knowledge proofs for NP. In Ernest F. Brickell, editor, *Advances in Cryptology – CRYPTO'92*, volume 740 of *Lecture Notes in Computer Science*, pages 215–227. Springer, August 1992.

[13] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. MiniLEGO: Efficient secure two-party computation from general assumptions. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 537–556. Springer, May 2013.

[14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled ram revisited. In PhongQ. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 405–422. Springer Berlin Heidelberg, 2014.

[15] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *EUROCRYPT*, 2014.

[16] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229. ACM Press, May 1987.

[17] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):691–729, 1991.

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

[19] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM*

CCS 12: 19th Conference on Computer and Communications Security, pages 513–524. ACM Press, October 2012.

[20] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. Cryptology ePrint Archive, Report 2007/155, 2007. http://eprint.iacr.org/2007/155.

[21] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J. Malozemoff. Amortizing garbled circuits. In *Advances in Cryptology – CRYPTO 2014.*, 2014.

[22] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, August 2003.

[23] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th Annual ACM Symposium on Theory of Computing*, pages 21–30. ACM Press, June 2007.

[24] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 955–966. ACM Press, November 2013.

[25] Seny Kamara, Payman Mohassel, and Ben Riva. Salus: a system for server-aided secure function evaluation. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12: 19th Conference on Computer and Communications Security*, pages 797–808. ACM Press, October 2012.

[26] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. Cryptology ePrint Archive, Report 2014/137, 2014. http://eprint.iacr.org/.

[27] Mehmet S. Kiraz and Berry Schoenmakers. A protocol issue for the malicious case of yao's garbled circuit construction, 2006. http://www.win.tue.nl/~berry/papers/wic06.pdf.

[28] Vladimir Kolesnikov and Ranjit Kumaresan. Improved secure two-party computation via information-theoretic garbled circuits. In Ivan Visconti and Roberto De Prisco, editors, *SCN 12: 8th International Conference on Security in Communication Networks*, volume 7485 of *Lecture Notes in Computer Science*, pages 205–221. Springer, September 2012.

[29] Vladimir Kolesnikov, Payman Mohassel, and Mike Rosulek. Flexor: Flexible garbling for xor gates that beats free-xor. In JuanA. Garay and Rosario Gennaro, editors, *Advances in Cryptology CRYPTO 2014*, volume 8617 of *Lecture Notes in Computer Science*, pages 440–457. Springer Berlin Heidelberg, 2014.

[30] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008: 35th International Colloquium on Automata, Languages and Programming, Part II*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, July 2008.

[31] Benjamin Kreuter, abhi shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 14–14. USENIX Association, 2012.

[32] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. Cryptology ePrint Archive, Report 2013/079, 2013. http://eprint.iacr.org/2013/079.

[33] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 1–17. Springer, August 2013.

[34] Yehuda Lindell and Benny Pinkas. A proof of yao's protocol for secure two-party computation. Cryptology ePrint Archive, Report 2004/175, 2004. http://eprint.iacr.org/2004/175.

[35] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 52–78. Springer, May 2007.

[36] Yehuda Lindell and Ben Riva. Cut-and-choose Yao-based secure computation in the online/offline and batch settings. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO (2)*, volume 8617 of *Lecture Notes in Computer Science*, pages 476–494. Springer, 2014.

[37] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating efficient RAM-model secure computation. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2014.

[38] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734. Springer, May 2013.

[39] Payman Mohassel and Ben Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 36–53. Springer, August 2013.

[40] Benjamin Mood, Debayan Gupta, Joan Feigenbaum, and Kevin Butler. Reuse It Or Lose It: More Efficient Secure Computation Through Reuse of Encrypted Values. In *ACM CCS*, 2014.

[41] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two party secure computation. Cryptology ePrint Archive, Report 2008/427, 2008. http://eprint.iacr.org/2008/427.

[42] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *TCC 2009: 6th Theory of Cryptography Conference*, volume 5444 of *Lecture Notes in Computer Science*, pages 368–386. Springer, March 2009.

[43] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, December 2009.

[44] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, April 1979.

[45] Manoj Prabhakaran, Alon Rosen, and Amit Sahai. Concurrent zero knowledge with logarithmic round-complexity. In *43rd Annual Symposium on Foundations of Computer Science*, pages 366–375. IEEE Computer Society Press, November 2002.

[46] Michael O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005. http://eprint.iacr.org/2005/187.

[47] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *ACM CCS 01: 8th Conference on Computer and Communications Security*, pages 196–205. ACM Press, November 2001.

[48] Mike Rosulek. The structure of secure multi-party computation. In *PHD thesis*. 2009. http://hdl.handle.net/2142/13698.

[49] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252. Springer, August 1989.

[50] abhi shelat and Chih-Hao Shen. Fast two-party secure computation with minimal assumptions. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 523–534. ACM Press, November 2013.

[51] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 197–214. Springer, December 2011.

[52] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13: 20th Conference on Computer and Communications Security*, pages 299–310. ACM Press, November 2013.

[53] Stefan Tillich and Nigel Smart. Circuits of Basic Functions Suitable For MPC and FHE. http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/.

[54] Stefan Wolf and Jürg Wullschleger. Oblivious transfer is symmetric. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 222–232. Springer, May / June 2006.

[55] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164. IEEE Computer Society Press, November 1982.

[56] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE Computer Society Press, October 1986.

APPENDICES

Appendix A: Streaming Cut-and-choose Protocol Efficiency

To signify the efficiency advantages of our streaming cut-and-choose protocol, we compare our approach with a naive but natural transformation of [19] from semi-honest to malicious security.

A.1 Naive Approach

In order to check consistency of the shared state values passing from one circuit to another, one could compute the one-time MAC of the shares in one circuit and verify the MACs in the next. And to maintain integrity and privacy of the memory blocks, a natural solution is to encrypt them using an authenticated encryption scheme and store a ciphertext that is decrypted whenever the memory location is accessed. Furthermore, one would need to repeat the cheating-recovery component after each timestep (or otherwise use $3s$ threads).

For the one-time MAC, we use the efficient scheme of [50] which we used earlier for input-consistency. This way, MACing is essentially free (since it is all XOR gates) while it costs only $2M$ AND gates to verify where M is the size of the input to the MAC. For the authenticated encryption (AE), one can use any standard AE scheme such as the efficient OCB-AES128 [47] which requires two AES calls when encrypting only one block of input. The decryption cost is similar with an extra τ AND gates where τ is the length of the authenticity tag (let $\tau = 128$). Assuming that an AES circuit implementation would require 6 800 non-XOR gates [53], authenticated encryption of a block of 128 bit would require a circuit size of 13 600 non-XOR gates while decryption requires 13 728 non-XOR gates.

In the construction of [19], the tree-ORAM circuit can be broken into four main circuits: 1) the circuit that given the shares of a virtual address, returns its corresponding label to the Receiver, 2) a circuit that given the shares of a virtual address and an encrypted path from root to a leaf, returns the shares of the data corresponding to the virtual address and removes it from the path, 3) a circuit that adds the removed data

to the root node, and 4) a circuit that given a label, evicts the nodes from root to that label. Note that one would need to apply a separate cheating recovery for each of these circuits.

Given these circuits, we compute the total number of bits (state information) that are passed between circuits. We also compute the number of times that we need to call encryption and decryption algorithms on the memory items. Note that these numbers are for a single ORAM operation.

Consider the following parameters. The number of actual data items stored in memory is denoted by N. In the level-0 tree of the ORAM, each node contains a constant number of blocks, Z. Each block consists of a metadata section of length D and a data section of the same size. Encrypting a block is implemented by AES-128. The security parameter (for key length and the length of the tag in authenticated encryption) is S'. We also denote the Sender side storage for the ORAM by CS. For simplicity, we consider the case of a non-recursive ORAM. Therefore, CS is equal to $N \times D$ (i.e. Sender needs to store his share of metadata for all memory locations). Since we are assuming the use of cheating recovery technique, the number of threads is $S = s$.

To compare the efficiency of our approach with the naive transformation, we compare the overhead incurred by each approach. The overhead is computed in three aspects: 1) the number of extra gates necessary, 2) the extra input consistency checks, and 3) the extra storage requirement on Sender's side. These extra cost are computed over the run-time (T) of the program. To clarify what we mean by "extra" overhead, consider the following.

If the size of a circuit (number of non-XOR gates) computing a semi-honest 2PC ORAM is denoted by SO and it stores CS bytes of data in Sender's side, using cut-and-choose and cheating recovery, we would at least need a circuit size of $MS = s \times SO$ for cut-and-choose and $3s \times |x_1|$ non-XOR gates for cheating recovery. We would also need $s \times CS$ bytes at Sender's side. Moreover, we would require the usual input consistency checks on x_1. Therefore, in the run-time of the program, we would need $MS_T = MS \times T + 3s \times |x_1|$ non-XOR gates and $CS_s = s \times CS$ bytes of storage. Any cost other than MS_T, CS_s and the input consistency checks on the $|x_1|$ is considered an overhead. In what follows, we compute the overhead of the naive transformation approach.

For each invocation of ORAM, we have the following costs. We need to apply MACing and verification for $8D + 2CS$ bits. The authenticated encryption and decryption are

each called on $3Z \log N + Z$ blocks. We need to check input consistency on $2D + 3S' + CS$ bits of data. And finally, the cost of cheating recovery for a circuit with input size M is $3s \times M$ non-XOR gates. Thus, for an ORAM application with running time T and assuming the use of cheating recovery, the overhead for time-steps t_1 to t_2 such that $t' = t_2 - t_1$ (corresponding to a single ORAM call) is as follows.

- MAcing: almost free.
- Verification: $t's \times (2 \times (8D + 2CS))$ non-XOR gates.
- Authenticated Encryption: $t's \times (13\,600 \times (3Z \log N + Z))$ non-XOR gates.
- Authenticated Decryption: $t's \times (13\,728 \times (3Z \log N + Z))$ non-XOR gates.
- Cheating Recovery: $3t's \times (8D + 2CS)$ non-XOR gates.

Note that during the run time of a program, many such ORAM calls are performed such that $T = t' \times num\_of\_calls$.

Given $D = 64$ (so that we can feed $2D = 128$ blocks of data to AES), $N = 2^{10}$, $S' = 128$, $s = 40$, $Z = 4$, and $CS = N \times D$ the total size of the overhead is $T \times 154.36 \times 2^{20}$ non-XOR gates. We would also have a computational overhead of $O(T \times IC \times ND)$ for input consistency checks, where IC is the overhead of input consistency check for one bit of data on s garbled circuits. The Sender storage does not have any overhead.

A.2 Our approach

In our approach, we do not need to check the correctness of the state information using MAC. We also, do not need authenticated encryption and decryption. Moreover, we perform the cheating recovery only once at the end of the protocol. Therefore, our only overhead is introduced by the final cheating recovery which is equal to $3s \times (|x_1| + 1)$ (see section 4.4.5), where x_1 is the input to the circuit in the first time-step. Notice that only $3s$ of it is considered "extra" overhead.

Our approach achieves the above at a cost of increasing the Sender's storage requirements. In our approach Sender needs know for each memory location and for each thread, which circuit updated that location (i.e. he needs to store the seed ($|seed| = S'$) of the circuit) and also when was the last update performed (i.e. he needs to store a

time-step t ($|t| = \log T$)). This results in an extra $N \times s \times (S' + \log T)$ storage for Sender. As for input consistency, note that we do not need any input consistency checks for the intermediate circuits which are responsible for ORAM access.

Given the same concrete parameters as above, with the addition of $|x_1| = 128$ the overheads are as follows. Our approach needs only 120 extra non-XOR gates at the cost of an extra $5MB + \log T \times 40KB$ of Sender storage.

Table A.1 provides a comparison of the overhead of the two approaches. Notice that as the running time increase our performance on circuit overhead increases linearly while the storage requirements increases only logarithmic. As can be seen in this table, our approach saves orders of magnitude on circuit size (number of non-XOR gates) and removing the need for costly input consistency checks, while adding only a small overhead on Sender storage size.

Table A.1: Comparison of "overhead" of naive implementation with streaming cut-and-choose approach

	Naive implementation	**Streaming cut-and-choose**
Circuit Size	$(T \times 154.36 \times 2^{20})$ non-XOR gates	(120) non-XOR gates
Sender Storage	0	$5MB + \log T \times 40KB$
Input Consistency Checks	$O(T \times IC \times ND)$	0

Appendix B: Concrete Bounds for Batch Preprocessing Protocol

Here we compute the number of circuits ρ needed per bucket in the protocol of Section 4.3. Let T denote the total number of time steps taken by the RAM program.

In that protocol, P_1 generates $2\rho T$ circuits and exactly half are checked. The remaining ones get placed randomly into T buckets of ρ circuits each.

Let $\mathbb{B}(\rho, T, m)$ denote the probability that some bucket contains a minority of good circuits, when m circuits are bad. Then we have the following recurrence:

$$\mathbb{B}(\rho, T, m) = \sum_{i=0}^{\rho} \frac{\binom{m}{i}\binom{\rho T - m}{\rho - i}}{\binom{\rho T}{\rho}} \left\{ \begin{array}{rl} \text{if } i < \rho/2 \text{ then} & \mathbb{B}(\rho, T-1, m-i) \\ \text{else} & 1 \end{array} \right\}$$

In this recurrence, i indexes the number of bad circuits in the first bucket. The fraction gives the probability of the first bucket receiving exactly i bad circuits. If $i < \rho/2$ then the condition is not yet met and it must further hold on the remaining $T-1$ buckets; if $i \geq \rho/2$ then the condition is met (hence 1).

Then let $\mathbb{B}^*(\rho, T, m)$ denote the overall probability that an adversary will be successful by generating m bad circuits. Since the bad circuits must survive the cut and choose, and then a minority-good bucket is generated, we have:

$$\mathbb{B}^*(\rho, T, m) = \frac{\binom{2\rho T - m}{\rho T}}{\binom{2\rho T}{\rho T}} \cdot \mathbb{B}(\rho, T, m)$$

A value of ρ is sufficient to achieve security 2^{-s} if we have

$$\max_m \{\mathbb{B}^*(\rho, T, m)\} < 2^{-s}$$

Using these recurrences, we were able to exactly compute the minimal values of ρ for $s = 40$ and selected values of T:

T	minimum ρ needed:
100	13
250	11
500	9
5,000	7
100,000	7
500,000	5

These are admittedly a very small sample size, though we can report that the points are fit closely ($r = 0.97$) by the linear regression $\rho = 1.86 \cdot (40/\log_2 T) + 1.46$.

We note that the analyses of [21] are slightly different, in that they need only a single good circuit in each bucket (i.e., the adversary succeeds only by making a bucket with no good circuits).

Checking a different fraction of circuits. In [36], it is suggested to check a different (i.e., not 1/2) fraction of circuits in the offline phase. Indeed, if the parties check a smaller fraction of circuits, then P_1 generates fewer circuits overall (in the offline phase) but P_2 evaluates more circuits *per timestep* in the online phase (i.e., buckets must be bigger).

Suppose that $1 - \phi$ fraction of circuits are checked in the offline phase. In order to have T buckets of ρ circuits each, P_1 must generate $N = \lceil \rho T/\phi \rceil$ circuits total and the parties must check $N - \rho T$ of them. Then the probability of m bad circuits surviving the cut and choose is:

$$\mathbb{B}^*(\rho, T, m) = \frac{\binom{\lceil \rho T/\phi \rceil - m}{\lceil \rho T/\phi \rceil - \rho T}}{\binom{\lceil \rho T/\phi \rceil}{\lceil \rho T/\phi \rceil - \rho T}} \cdot \mathbb{B}(\rho, T, m)$$

Following [36], we compute the parameters for several values of T and ϕ (again for $s = 40$):

T	fraction checked $(1-\phi)$	circuits per timestep	
		online only (bucket size)	total (eval + check)
100	0.80	9	45.0
100	0.60	11	27.5
100	0.40	13	21.7
100	0.25	15	20.0
500	0.90	7	70.0
500	0.50	9	18.0
500	0.25	11	14.7
1000	0.80	7	35.0
1000	0.40	9	15.0
1000	0.20	11	13.7
5000	0.50	7	14.0
5000	0.15	9	10.6
10^4	0.35	7	10.8
10^4	0.10	9	10.0
5×10^4	0.15	7	8.2

We note that [36] also prove a bound on the bucket size ρ; namely, if:

$$\rho \geq \frac{2s + 2\log T - \log(-1.25 \log \phi) - 1}{\log T + \log(-1.25 \log \phi) - 2}$$

then the total probability of a *majority-bad* bucket is at most 2^{-s}, when using buckets of size ρ. However, the exact bounds that we have computed are significantly tighter.