Collusion-Resistant Functional Encryption for RAMs

Prabhanjan Ananth[1], Kai-Min Chung[2], Xiong Fan[3] and Luowen Qian[4]

[1] UC Santa Barbara, Santa Barbara, CA, USA, prabhanjan@cs.ucsb.edu
[2] Academia Sinica, Taipei, Taiwan, kmchung@iis.sinica.edu.tw
[3] Rutgers University, Piscataway, NJ, USA, xiong.fan@rutgers.edu
[4] Boston University, Boston, MA, USA, luowenq@bu.edu

Abstract. In recent years, functional encryption (FE) has established itself as one of the fundamental primitives in cryptography. The choice of model of computation to represent the functions associated with the functional keys plays a critical role in the complexity of the algorithms of an FE scheme. Historically, the functions are represented as circuits. However, this results in the decryption time of the FE scheme growing proportional to not only the worst case running time of the function but also the size of the input, which in many applications can be quite large.

In this work, we present the first construction of a public-key collusion-resistant FE scheme, where the functions, associated with the keys, are represented as random access machines (RAMs). We base the security of our construction on the existence of: (i) public-key collusion-resistant FE for circuits and, (ii) public-key doubly-efficient private-information retrieval [Boyle et al., Canetti et al., TCC 2017]. Our scheme enjoys many nice efficiency properties, including input-specific decryption time.

We also show how to achieve FE for RAMs in the bounded-key setting with weaker efficiency guarantees from laconic oblivious transfer, which can be based on standard cryptographic assumptions. En route to achieving our result, we present conceptually simpler constructions of succinct garbling for RAMs [Canetti et al., Chen et al., ITCS 2016] from weaker assumptions.

Keywords: Functional Encryption, RAMs

1 Introduction

Functional Encryption. In the recent years, several interesting cryptographic primitives have been proposed in the domain of computing on encrypted data, with one such primitive being *functional encryption* [11,51,52]. This notion allows for an entity to encrypt their input x such that anyone in possession of secret keys associated with functions $f_1, \ldots, f_q$, also referred to as functional keys, can decrypt this ciphertext to obtain the values $f_1(x), \ldots, f_q(x)$ and nothing else. The setting where q is not a priori bounded is called the collusion resistant setting and will be the primary focus of this work.

Functional encryption (FE) has proven to be a useful abstraction for many theoretical applications, including constructing indistinguishability obfuscation [5,10], succinct randomized encodings [1,6,34], watermarking schemes [40], proving lower bounds in differential privacy [47], proving hardness of finding a Nash equilibrium [9,32] and many more.

Model of Computation. A vast majority of FE constructions model the functions associated with the functional keys as circuits. While circuits are easy to work with, when compared to other models of computation, they come with many disadvantages. The parameters in the system tend to grow *polynomially in the worst-case time bound* of the

function; this includes the decryption time. Even worse, for functions that take sub-linear runtime in the "big data" setting, the decryption time would now take time proportional to the size of the entire data, which could be massive.

Designing FE for Alternate Models of Computation. These drawbacks prompt us to look beyond circuits and construct FE for more general models of computation. One general model of computation that we could hope to support is random access machines (RAMs). There are many advantages to FE for RAMs, we will mention a couple of them now and defer more when we formally define the primitive in the next section: firstly, the parameters of the scheme do not grow with the worst-case time bound and moreover, the decryption time is input-specific.

Despite its utility, the feasibility of collusion-resistant FE for RAMs had not been explored in prior works. Prior works did make partial progress in this direction by either considering weaker models of computation such as finite automata [2], Turing machines [1, 6, 7, 34, 45] or in the single-key setting [36]*. However, the problem of constructing FE for RAMs was unanswered and has been one of the important open problems in this area.

1.1 Contributions

We resolve this open problem; we give the first feasibility result of functional encryption for RAMs. Before stating our result, we first elaborate on the definition of FE for RAMs. A public-key functional encryption for RAMs consists of the following algorithms:

- The setup algorithm Setup that produces a public key pk and a master secret key MSK. The runtime of the setup algorithm is polynomial in λ (security parameter) and grows poly-logarithmically in the worst-case runtime bound T.

- The key generation algorithm KeyGen that takes as input MSK, a RAM program P and outputs a functional key for P, denoted by sk_P. The running time of key generation is only proportional to λ, the description size of P and grows poly-logarithmically in T.

- The encryption procedure Enc takes as input MSK, database D and outputs a ciphertext CT. The running time of the encryption procedure grows polynomially in λ, $|D|$ and poly-logarithmically in T.

- The decryption procedure Dec, modeled as a RAM program, takes as input ciphertext CT, functional key sk_P and produces the output $P^D()$. The runtime of decryption should grow proportional only to t and λ, where t is the time to execute P^D.

The security notion† for the above notion can be appropriately defined along the same lines as (collusion-resistant) FE for circuits.

In terms of efficiency, FE for RAMs schemes enjoy better efficiency guarantees than FE for circuits schemes in terms of both the running time of the key generation algorithm as well as the running time of the decryption algorithm. We clarify this in Figure 1.

*Note that the work of [36] also construct an FE for RAMs scheme in the bounded-key setting: however, the decryption time of the bounded-key FE scheme grows polynomially in the database size and thus doesn't enjoy the sublinear decryption runtime property that we desire.

†The security notion we consider in this work is indistinguishability-based (IND-based) selective security. We delve more on this when we formally define FE for RAMs in the technical sections.

‡A well-known technique for decreasing the running time from T to t is to issue $\log T$ decryption keys, with the i-th one running in time at most 2^i.

	FE for Circuits	**Our work**
RunTime(Setup)	$\mathsf{poly}(\lambda)$	$\mathsf{poly}(\lambda)$
RunTime(KeyGen)	$\mathsf{poly}(\lambda, \lvert P\rvert, \lvert\mathbf{D}\rvert, \mathbf{T})$	$\mathsf{poly}(\lambda, \lvert P\rvert)$
RunTime(Enc)	$\mathsf{poly}(\lambda, \lvert D\rvert)$	$\mathsf{poly}(\lambda, \lvert D\rvert)$
RunTime(Dec)	$\mathsf{poly}(\lambda, \lvert\mathbf{D}\rvert, t)^{\ddagger}$	$\mathsf{poly}(\lambda, t)$

Figure 1: Comparison of efficiency guarantees of FE for circuits via naively simulating RAM programs (that is, to issue a key for a program P and time bound T, generate a key for a circuit that runs P for T time steps) and our work. We denote P to be the program input to the key generation algorithm, D to be the database input to the encryption algorithm and T to be the worst case running time of P. We denote t to be the running time of P on D. Since, the typical setting of T is 2^{λ}, we omit mentioning the dependence on poly-log factors in T.

Main Result: Collusion-resistant FE for RAMs. We show how to generically transform any (collusion-resistant) FE for circuits scheme into a (collusion-resistant) FE for RAMs scheme. Our transformation additionally assumes the existence of public-key doubly-efficient private information retrieval (PK-DEPIR) scheme, introduced independently by the works of Boyle et al. [16] and Canetti et al. [21].

In more detail, we show the following.

Theorem 1 (Informal)**.** *There exists a collusion-resistant public-key FE scheme for* RAMs *assuming the existence of:*

- *collusion-resistant public-key FE for circuits and,*
- *public-key doubly efficient PIR [16, 21].*

We note that the construction of public-key DEPIR is currently based on security of VBB for specific class of circuits. However, we note that even demonstrating the feasibility of FE for RAMs from *any* cryptographic assumption was wide open. Thus, we believe that our work takes an important step towards establishing the feasibility of FE for RAMs. We point out that a related primitive, FHE for RAMs [41], was also based on the assumption of public-key DEPIR.

Our construction involves a novel combination of pebbling techniques [31], rewindable ORAMs [41], and hybrid functional encryption techniques [3]. We only work in the selective security setting, where the challenge message query needs to be declared by the adversary even before looking at the public key.

Observe that the assumption of FE for circuits is inherent in Theorem 1 since FE for RAMs imply FE for circuits. It is natural to ask whether the assumption of public-key DEPIR is inherent. While we don't answer this question, we still make a useful observation: an FE for RAMs scheme implies a weaker notion, called *secret-key* DEPIR.

Theorem 2 (Informal)**.** *Assuming the existence of unbounded private-key FE for RAMs, there exists a construction for unbounded secret-key DEPIR.*

The works of Boyle et al, Canetti et al [16, 21] also proposed constructions for secret-key doubly efficient PIR; while they are based on new cryptographic assumptions, a thorough study of the assumptions was recently conducted by [15].

Intermediate Result: Succinct Garbled RAMs from Falsifiable Assumptions. Towards proving our main result, we obtain a new construction[¶] of succinct garbled RAMs [8, 19,

[¶]In fact, we define a stronger version called succinct *reusable* garbled RAM; this notion implies succinct garbled RAM.

20, 23, 46]. A succinct garbling scheme for RAMs consists of the following algorithms: (i) Database encoding algorithm that encodes a database D in time $\mathsf{poly}(\lambda, |D|)$, (ii) RAM garbling algorithm garbles a program P in time $\mathsf{poly}(\lambda, |P|)$ and, (iii) Evaluation algorithm that takes as input garbling of D, garbling of a program P and outputs $P^D()$, in time polynomial in $(\lambda, |P|, |D|, t)$, where t is the running time of $P^D()$.

It has two advantages over prior constructions: (i) first, it is arguably simpler than existing constructions [4, 18, 19, 23] and, (ii) second, it is based on polynomially secure functional encryption scheme for circuits (a falsifiable assumption) as opposed to existing constructions which are based on indistinguishability obfuscation[‖] schemes (a non falsifiable assumption).

Formally, we prove the following.

Theorem 3 (Informal)**.** *There exists a succinct garbling scheme for RAMs assuming polynomially secure (collusion-resistant) public-key functional encryption for circuits.*

Bounded-Key FE for RAMs. Our techniques also extend naturally to the bounded-key setting. In this setting, the adversary can only query an a priori bounded number of functions in the security experiment. We show how to construct a bounded-key FE for RAMs from standard assumptions; unfortunately, the resulting FE for RAMs scheme does not enjoy the same efficiency properties as before. In particular, the algorithms run in time polynomial in the worst case time bound. Nonetheless, this still performs better than the bounded key FE for circuits scheme since the decryption time only grows with the worst case time bound and in particular, does not explicitly depend on the size of the database encrypted. Formally,

Theorem 4 (Informal)**.** *Assuming the existence of laconic oblivious transfer [24] and public-key encryption, there exists a bounded-key public-key FE for RAMs scheme satisfying the following efficiency properties:*

- *The time to compute setup is* $\mathsf{poly}(\lambda, Q, |P|, T)$*, where* T *is the worst case time bound and* Q *is the collusion bound.*
- *The time to compute the key generation of a program* P *is* $\mathsf{poly}(\lambda, Q, |P|, T)$*.*
- *The time to compute the encryption of a database* D *is* $\mathsf{poly}(\lambda, Q, |P|, |D|, T)$*.*
- *The time to compute the decryption of a functional key associated with* P *and a ciphertext of database* D *is* $\mathsf{poly}(\lambda, Q, |P|, t)$*, where* t *is the runtime of* $P^D()$*.*

In comparison, a bounded key FE for circuits scheme has similar setup, key generation and encryption runtimes except that the decryption time is polynomial in $(\lambda, Q, |D|, |P|, t)$. When $t \ll |D|$, our bounded key FE for RAMs scheme outperforms bounded key FE for circuits schemes.

The primitive of laconic oblivious transfer can be instantiated using a host of well studied assumptions (for example, computational Diffie-Helman (CDH), learning with errors [17, 24]). Thus, we obtain different constructions of bounded-key FE for RAMs based on standard assumptions.

Corollary 5 (Informal)**.** *Assuming* $\mathcal{X} \in \{$*CDH,LWE,Factoring*$\}$*, there exists a bounded-key public-key encryption scheme for RAMs.*

[‖] In the technical sections, we use indistinguishability obfuscation for circuits with logarithmic inputs to construct succinct reusable garbled RAMs. However, it has been shown [49] that iO for logarithmic inputs is equivalent to collusion-resistant functional encryption for circuits.

Related Work. Goldreich and Ostrovsky [38] initiated the area of building cryptographic primitives for RAM programs and since then, several works have proposed cryptographic constructions for RAM computations: for example, garbling schemes [4, 8, 18–20, 23, 29, 30], secure multiparty computation for RAMs [28, 43], doubly-efficient private-information retrieval [16, 21], private anonymous data access [42] and fully homomorphic encryption for RAMs [41]. Of particular interest to us is the work of Gentry et al. [36] which introduced and constructed (single-input) functional encryption for RAMs in the single-key setting. We view our work as continuing this exciting line of research.

2 Technical Overview

We present an overview of our construction.

Recap: Garbled RAMs. Towards building FE for RAMs, we first start with a weaker but similar notion of FE for RAMs, popularly referred to as garbled RAMs [29, 30, 35] in the literature. A garbled RAM allows for separately encoding a RAM program-database pair (P, D) such that the encodings only leak the output $P^D()$ (here we assume the program input is hardcoded in the program); computing both the encodings requires a private key that is not revealed to the adversary. Notice that a garbled RAM scheme already implies a *one-time, secret key* FE for RAM scheme; meaning that the adversary only gets to make a single ciphertext query and a single functional key query in the security experiment.

Traditionally, the following two-step approach is employed to construct a garbled RAM scheme:

- First construct a garbled RAM scheme in the UMA (unrestricted memory access) setting; the setting where the memory access pattern is not hidden.
- To hide the access pattern, generically combine any garbled RAM scheme satifying UMA security with an oblivious RAM scheme [38].

The blueprint employed to construct a garbled RAM scheme in the UMA setting is the following: to garble a RAM program P (associated with a step circuit C), database D, generate T garbled circuits [54], where T is an upper bound on the running time of P. The i^{th} garbled circuit performs the "CPU circuit" which evaluates the i^{th} time step of P. The garbling of P consists of all T garbled circuits.

To evaluate a garbling of P on a suitably encoded database D, perform the following operations for $i = 1, \ldots, T-1$: evaluate the i^{th} garbled circuit to obtain output encodings of the i^{th} step of execution of P^D. Next, we compute the *recoding step* that converts the output encodings of the i^{th} step into the wire labels for the $(i+1)^{th}$ garbled circuit; only the recoding step involves the encoded database where we retrieve information and enforce honest evaluation. The resulting wire labels will be used to evaluate the $(i+1)^{th}$ garbled circuit.

The output of the T^{th} garbled circuit is the output of execution of P^D.

Recall that in the UMA setting, we do not hide memory access pattern, memory content, or intermediate states. In order to achieve full security, we additionally need to compile the original program with additional protection, usually this involves a specially crafted oblivious RAM scheme to hide the access pattern, and a suitable secret key encryption to hide the rest.

Towards FE for RAMs: Challenges. To leap from a toy case of FE for RAMs, a.k.a. garbled RAMs, to building a full-fledged collusion-resistant public-key FE for RAMs involves many hurdles. We start by highlighting two such challenges.

Challenge: Parallel** Reusability. Let the adversary receive as input, encryption of a challenge database D^* and functional keys $\mathsf{sk}_{P_1}, \ldots, \mathsf{sk}_{P_q}$ associated with RAM programs $P_1, \ldots, P_q$. We can decrypt the *same* encryption of D^* using the different functional keys $\mathsf{sk}_{P_1}, \ldots, \mathsf{sk}_{P_q}$ to obtain $P_1^{D^*}, \ldots, P_q^{D^*}$.

Typically, in the RAM setting, however, reusability has only been studied in the sequential setting (also called persistent memory setting [35]) where P_1 first acts on D^* to obtain an updated database; P_2 then acts upon the updated database and so on. To construct FE for RAM, the notion of parallel reusability is required, where different programs $P_1, \ldots, P_q$ need to act upon the same initial database D^*.

Prior results show that some of the existing garbled RAMs are insecure in the parallel reusability setting [42]††.

Challenge: Succinctness. Recall that we enforce stringent efficiency requirements on FE for RAMs schemes: the parameters should neither grow with the database length nor with the worst-case time bound, the decryption time should only grow proportional to the input-specific running time and so on. Even for simpler primitives such as randomized encodings, achieving succinctness has proven to be very challenging; for instance, the constructions of *succinct* garbled RAMs by [19, 23] are quite complex and involve heavy tools.

Moreover, unlike weaker models, generic constructions of FE using succinct garbling do not work in the RAM setting. For instance, in the setting of Turing machines, here is an approach to obtain FE for Turing machines from FE for circuits: use FE for circuits to generate a succinct garbling of the database encrypted and the TM associated with the functional key. Such solutions would necessarily blow up the decryption time proportional to the size of the database encrypted, even if the program only runs in sublinear time.

Known Tools. The above two challenges are not new and have presented themselves in different contexts. We mention some of the relevant contexts below.

Succinct Garbling for RAMs [8, 19, 20, 23]: Succinct garbling schemes for RAMs do solve the problem of succinctness but does not satisfy the parallel reusability property. They either only allow the evaluation of one garbled program, or only allow evaluating several programs sequentially in a stateful manner, while for functional encryption we would like the program evaluation to be stateless.

FE for circuits [11, 51, 52]: As we mention in the introduction, FE schemes for circuits do address the challenge of parallel reusability; functional keys associated with programs $P_1, \ldots, P_Q$ can be used in parallel to decrypt an encryption of x. However they do not achieve succinctness since the decryption time grows with the worst-case runtime of the computation.

Rewindable ORAMs [42]: A recently introduced primitive, rewindable ORAM, allows for rewinding the encoded database of the ORAM scheme to an earlier state. The security property states that the access patterns generated even after rewinding the encoded database should not reveal any information about the underlying database. This primitive does address the challenge of parallel reusability, succinctness (only a small amount of secret state needed to perform evaluation) but in itself is not useful since this gives an interactive solution and hence needs to be used in conjunction with other (possibly non-interactive) primitives.

††To be precise, [42] shows that traditional ORAM schemes are insecure in the parallel reusability setting. This correspondingly means that the garbled RAMs schemes building upon these ORAM schemes would correspondingly be insecure in the parallel setting.

2.1 Our Template

We show how to combine the techniques used to construct the above seemingly unrelated tools to obtain a construction of FE for RAMs. As mentioned earlier, the current known constructions of succinct garbling schemes for RAMs are difficult to work with. We will first simplify (and improve!) these constructions before achieving our main result.

The template for the rest of the overview is as follows:

- We first tackle the challenge of succinctness. We present a new construction of a garbled RAM (GRAM) scheme. This will serve as an alternative to existing schemes which are significantly more complex and additionally assumes sub-exponentially secure FE for circuits . Our scheme is simpler and only assumes polynomially-secure FE for circuits.

- We upgrade this succinct GRAM scheme to satisfy parallel reusability; the same garbled database can be evaluated upon by multiple garbled programs. We call this succinct reusable GRAM. This notion would imply a single-ciphertext collusion-resistant FE for RAMs in the secret-key setting. The adversary can only make a single ciphertext query. One of the important tools we use to achieve parallel reusability is rewindable ORAMs.
  In the technical sections, we present the construction of succinct reusable GRAM directly, instead of first presenting the non-reusable version and then upgrading it to the reusable version. We present the upgrading step in this overview to explain the construction better to the reader.

- Finally, we combine succinct reusable GRAMs with collusion-resistant FE for *circuits* to obtain collusion-resistant FE for RAMs.

2.2 Starting Point: Simpler, Better and Modular Succinct GRAM

Our starting point is the following template introduced by [8] to construct succinct garbled RAMs.

- We start with a *non-succinct* garbled RAM scheme, i.e. the parameters in the scheme could grow proportional to the worst runtime bound T of the computation. However, we still require that the evaluation runs in time proportional to the runtime of the computation and in particular, could be independent of the database length. Such a garbled scheme can be constructed from one-way functions [29–31], and these constructions follow the two-step approach that we have outlined at the beginning of the section.

- To go from a non-succinct to a succinct garbled RAM scheme, we need to reduce the size of the garbled program to be independent of the worst case bound T. We achieve this size reduction using program obfuscation[‡‡]. Specifically, we use obfuscation to delegate the execution of the non-succinct program garbling procedure to the time of evaluation. That is, to garble a program P via a succinct garbling scheme, compute an obfuscated circuit that produces a non-succinct garbling of P.

To make the above high level approach work, we need to nail down the precise properties that we need from the underlying non-succinct garbled RAM scheme. For starters, just obfuscating the non-succinct garbling procedure would not work: the size of the obfuscated circuit will be as large as the size of the non-succinct garbled program and thus, we didn't achieve size reduction.

[‡‡]A program obfuscation is a compiler that transforms a program P into a functionally equivalent program that hides all the implementation details of the original program. In the technical sections, we use a specific definition of obfuscation, called indistinguishability obfuscation.

Thus, we need to start with a non-succinct garbling scheme where the garbled program can be decomposed into many components such that the obfuscated circuit produces one component at at time. Even if we do this, arguing proof turns out to be tricky: a naive approach to reduce to the security of the non-succinct garbling scheme involves hardwiring the entire garbled program inside the obfuscated circuit but this again is not possible as it violates succinctness.

Local Simulatability: These issues are not unique to our setting and have already been encountered while designing succinct garbled RAMs with bounded space [8] or succinct garbled Turing machines [6,33]. They identified two main properties that are necessary for the underlying non-succinct garbling scheme to satisfy.

- The program being garbled can be broken down into small components (say, of size $\mathsf{poly}(\lambda, \log T)$) and each of these components can be garbled independently. This property also helps in proving security of the succinct garbled Turing machine without having to hardwire the entire garbled circuit inside the obfuscated circuit.
- The security proof of the non-succinct scheme should be argued in such a way that only a "small" (say, $\mathsf{poly}(\lambda, \log T)$) subset of the garbled program components need to be changed from one hybrid to the next hybrid.

We now revisit the template mentioned above and change the circuit being obfuscated to output the (non-succinct) garbled program, one component at a time. On input i, the obfuscated circuit outputs the i^{th} component of the garbled program, instead of producing the whole garbled program at once. To argue security, we carry out the hybrids of the non-succinct garbling scheme by only hardwiring a small subset of components at a time. By local simulatability, we are guaranteed that in each hybrid, the amount of hardwired information is never too large and therefore we achieve succinctness.

Therefore, we have reduced the problem of constructing succinct GRAM to identify and instantiate an appropriate non-succinct garbling scheme satisfying the above two properties. This is where previous works fall short. Their instantiations yielded succinct garbling schemes only for Turing machines [6,33] or succinct garbled RAMs with bounded space [8].

Non-Succinct Garbled RAMs with Local Simulatability[§§]: To construct (non-succinct) garbled RAM satisfying the local simulatability property, we split the construction into two parts: in the first part we construct a succinct garbled RAM with unprotected memory access (UMA), where we forget about protecting memory contents, access patterns and intermediate CPU states; in the second part, we bootstrap UMA-GRAM to fully secure GRAM.

For the first step, we observe that the UMA-secure adaptive garbled RAM construction of [31] already satisfies the local simulatability property. For the second part, previous schemes usually employ an ORAM to hide the memory access pattern and an encryption scheme to hide the memory content. However, these tools are not quite compatible with the local simulatability property, therefore, their compatible versions of ORAM with strong localized randomness, and timed encryption scheme – originally introduced by the same paper [31] to construct adaptive garbled RAMs – are needed for the proof.

Timed encryption, at a high level, is an encryption scheme that allows issuing encryption/decryption keys with growing power as the evaluation goes on, i.e. a key issued at time t can decrypt anything that was encrypted under time $t' \leq t$, but any message encrypted at a later time remains hidden. Using the tool of timed encryption allows us to use a sequence of hybrids to remove the timed encryption keys one by one (and hence allowing us to simulate each evaluation step *locally*), from the strongest (which is one

[§§]The terminology of local simulation is only introduced for the benefit of describing our techniques and will be implicit in our security proof.

hardwired in the last step circuit) to the weakest (which is the one hardwired in the first step circuit).

Looking ahead, there is another more subtle issue for constructing succinct GRAM that is not captured by local simulatability: in the succinct garbling scheme, we can only use a very small amount of randomness in the simulator, as otherwise the size of the simulated circuit will blow up and break succinctness. In particular, this means that we cannot simply hardcode the timed encryption of 0. For this issue, we develop timed encryption with *pseudorandom ciphertexts*, which is a timed encryption whose ciphertext is indistinguishable from uniformly random bitstrings; and construct it from one-way functions. Once we have that, we can simply use a PRF to generate all the simulated ciphertexts in a succinct way.

We now move on to hiding access pattern in a local simulatable way. Strong localized randomness property for ORAM, at a high level, simply requires that the randomness used by ORAM is equipped with some structural properties that will allow us to equivocate (and change) the randomness in a *local* way. For now, the ORAM with strong localized randomness constructed in [31] suffices for succinct (non-resuable) garbled RAM.

2.3 Succinct Garbled RAM: Achieving Reusability

Succinct GRAM alone itself is not going to be sufficient to construct FE for RAMs. Instead, it turns out to require the *reusability* property: given an encoding of a database D and multiple garbled programs $\widetilde{P}_1, \ldots, \widetilde{P}_q$, the adversary can recover the outputs $P_1^D(), \ldots, P_q^D()$ and moreover, the database encoding and the garbled programs do not leak any information about D beyond the outputs that can be recovered. We call this notion succinct *reusable* garbled RAM.

Note that this definition is different from the persistent memory setting [35]; the programs *sequentially* evaluate on the databases as against the parallel execution that we desire. In addition, we also require that the reusable GRAM also satisfies succinctness properties as defined in a succinct GRAM scheme.

From Succinct GRAM to Succinct Reusable GRAM. To construct a succinct reusable garbled RAM, again it is helpful to split things into two part: in the first part we construct a succinct *reusable* garbled RAM with unprotected memory access (UMA), and in the second part we use this UMA primitive to construct fully secure succinct *reusable* garbled RAM. Note that in UMA setting, essentially all we are protecting is the program execution, and we do not face much trouble in adapting the scheme above into the reusable setting. Therefore, we focus on the full security setting and highlight the new challenges in the reusability setting.

Challenges in Protecting Memory Content: To protect the content of the memory, we need to include the encryption key into our garbled program. However, once we have given out one garbled program, we can no longer invoke the security of the encryption scheme to say that the adversary has no information about the underlying database, as the garbled program contains a hardwired secret key. Indeed, the adversary can simply read from the encrypted database by simply reading the output of the garbled program. Therefore we need to remove the encryption keys in the hybrids very carefully. In the non-reusable setting, it has been shown in prior work [31] that using timed encryption fixes this issue. On a high level, their idea is to remove the encryption key one by one in each hybrid, in particular, they would remove the encryption key from the last garbled program (and write junk to the database instead) indistinguishably in the first hybrid, and then move forward and remove the encryption key in the second last garbled program, and so on. Essentially, timed encryption allows us to encrypt messages under a different key in each time step, while the decryption key can only decrypt messages before the

current timestep but not after, which allows the hybrid argument to go through. However, this security proof does not work in the reusable case: when we try to equivocate the output/database writes and remove the encryption key, the adversary could in principle still be able to distinguish the two distributions as the same timed encryption key still appears in other garbled programs.

In order to tackle this issue, we employ a different time step labeling and also a different hybrid strategy. In particular, instead of the time steps increasing in each garbled program, each garbled program will use a shared global time counter. Note that this also makes sense from the reusability point of view, as the evaluator can in principle evaluate garbled programs on the garbled database in any order that he wishes.

Now suppose we want to remove the strongest encryption key in the last step circuit. We can employ the following hybrid sequence: first, we use the security of UMA-GRAM to change each last step circuit into a dummy circuit that directly outputs the output in *all* garbled programs *in parallel* (to do it more carefully, we replace each garbled program one by one and argue each change is indistinguishable) – this effectively removes all the timed encryption keys that are used in the last time step; this allows us to do the next step which is to change the encrypted CPU states and write data into garbage *in parallel*; finally, we reverse the change of dummy circuit again in parallel. By doing so, we remove the strongest timed encryption key in *all* garbled programs at once. We can repeat this process for each remaining encryption keys until all encryption keys are removed from garbled program, at which point we can replace the database with an empty database and arrive at the simulated distribution.

Challenges in Protecting Memory Access Pattern: Another issue is that we need to protect the database read/write patterns in a way that is compatible with succinct UMA GRAM. Basically, we need to change each database read/write pattern without hardwiring too much additional information, which would blow up the size of the garbled program and break succinctness. This is further complicated by the fact that the adversary can evaluate different programs on the same database *in parallel* and compare the results to acquire additional information.

To resolve both these issues, we design a rewindable ORAM scheme satisfying strong localized randomness property. The starting point of the construction is the plain rewindable ORAM scheme given in [41], which consists of two parts: a read-only rewindable ORAM and a read-write non-rewindable ORAM. The idea of the construction is that the read-write ORAM will act as a read-write cache to the underlying database, which is encoded in the read-only ORAM.

Given this beautiful construction, it is straightforward to construct a rewindable ORAM scheme with strong localized randomness. In particular, we simply instantiate the read-write ORAM with the ORAM with strong localized randomness property. The access pattern in read-only ORAM is by definition locally sampled, and we can simulate the access pattern in read-write ORAM locally by using the strong localized randomness property of the read-write ORAM that we use.

2.4 Bootstrapping Step: From FE for Circuits to FE for RAMs

Once we construct a succinct reusable garbled RAM scheme, we show how to bootstrap a FE for circuits scheme into a FE for RAMs scheme. Our transformation is inspired by a similar transformation described in [26].

- To encrypt a database D, encode D using a succinct reusable GRAM scheme. Denote the output by $(\widetilde{D}, sk)$. Encrypt sk using an FE for circuits scheme; call the resulting

ciphertext ct. Output the ciphertext of the FE for RAMs scheme, $\mathsf{CT} = (\widetilde{D}, ct)$.

- To generate a functional key for a program P, generate a FE key for a circuit G that takes as input a secret key sk and produces a garbling of the program P with respect to sk; call the FE key SK_G. Set the functional key for the FE for RAMs scheme to be SK_G.
- The decryption algorithm first recovers the garbled program $\widetilde{P}$ by running the FE decryption algorithm. It then runs the succinct GRAM evaluation of $\widetilde{P}$ on $\widetilde{D}$ to obtain P^D.

To argue security, we can use the hybrid functional encryption technique of [3, 22] to first hardwire the garbled programs in the function keys and then invoke the reusable security of the GRAM scheme to prove the indistinguishability security of the FE scheme.

2.5 Organization

We organize the technical sections of our paper as follows:

- In Section 3, we introduce our notations and preliminaries, with additional preliminaries described in the full version.
- In Section 4, we present a construction of succinct reusable garbled RAM.

  First, we present the definition of succinct reusable garbled RAM in section 5.1. Next, in Section 5.2, we present a construction of succinct garbled RAM in the UMA setting. In this step, we use pebbling techniques in conjunction with indistinguishability obfuscation for inputs of logarithmic length (implied by functional encryption). Finally, in Section 5.3, we show how to transform UMA-secure garbled RAM to fully secure garbled RAM in the reusability setting. As a result, we obtain the construction of succinct reusable garbled RAM. We use the tool of rewindable ORAM in this step. The missing proofs in Section 5 are presented in the full version.
- In Section 6, we show how to combine (collusion-resistant) FE for circuits with succinct reusable garbled RAM to achieve (collusion-resistant) FE for RAMs. The missing proofs in Section 6 are presented in the full version. At last, we show implication of FE for RAMs to secret-key DEPIR in the full version as well.

3 Preliminaries

We denote λ to be the security parameter. We denote the computational indistinguishability of two distributions D_1 and D_2 by $D_1 \approx D_2$. We use the abbreviation PPT to denote probabilistic polynomial time algorithms. Additional preliminaries are presented in the full version.

RAM model of computation. We recall the definition of RAM computations. A RAM computation consists of a RAM program P and a database D. The representation size of P is independent of the length of the database D. The program P has random access to the database D. We denote the output to be P^D In more detail, the computation proceeds as follows.

The RAM program P is represented as a step-circuit C. It takes as input internal state from the previous step, location to be read, value at that location and it outputs the new state, location to be written into, value to be written and the next location to be read. More formally, for every $\tau \in T$, where T is an upper bound on the running time,

$$(\mathsf{st}^\tau, \mathsf{rd}^\tau, \mathsf{wt}^\tau, \mathsf{wb}^\tau) \leftarrow C(\mathsf{st}^{\tau-1}, \mathsf{rd}^{\tau-1}, b^\tau)$$

where we have the following:

- $\mathsf{st}^{\tau-1}$ denotes the state in the $(\tau-1)^{th}$ step and st^{τ} denotes the state in the τ^{th} step.
- $\mathsf{rd}^{\tau-1}$ denotes the location to be read from, as output by the $(\tau-1)^{th}$ step.
- b^{τ} denotes the bit at the location $\mathsf{rd}^{\tau-1}$.
- rd^{τ} denotes the location to be read from, in the τ^{th} step.
- wt^{τ} denotes the location to be written into in the τ^{th} step.
- wb^{τ} denotes the value to be written at τ-th step at the location wt^{τ}.

Remark 1. *(Additional Input)* In the literature, when defining RAM programs, we also additionally define an input x and the program in addition to having random access to D, takes as input x, and outputs P^D. Without loss of generality, we assume that the input x is part of the database and hence we omit including this as an explicit input to P.

Remark 2. *(Outputs)* In this work, we only consider RAM programs with boolean outputs. We can suitably extend the schemes we construct to handle multiple outputs at the cost of blowing up the parameters proportional to the output length.

3.1 Puncturable PRF

Puncturable PRFs [12, 14, 44] are PRFs for which a key can be given out such that, it allows evaluation of the PRF on all inputs, except for any polynomial-size set of inputs. The following definition is adapted from [53].

Definition 1 (Puncturable PRF)**.** A puncturable family of PRFs F mapping is given by a typle of ppt algorithms $(\mathsf{Gen}_F, \mathsf{Eval}_F, \mathsf{Punc}_F)$ and a pair of computable functions $n(\cdot)$ and $m(\cdot)$, satisfying the following conditions:

- **Functionality preserved under puncturing**: For every ppt adversary $\mathcal{A}$ such that $\mathcal{A}(1^\lambda)$ outputs a set $S \subseteq \{0,1\}^{n(\lambda)}$, then for all $x \in \{0,1\}^{n(\lambda)}$ where $x \notin S$, we have that

$$\mathsf{Pr}[\mathsf{Eval}_F(K,x) = \mathsf{Eval}_F(K_S,x) : K \leftarrow \mathsf{Gen}_F(1^\lambda), K_S = \mathsf{Punc}_F(K,S)] = 1$$

- **Pseudorandom at punctured points**: Foe every ppt adversary $(\mathcal{A}_1, \mathcal{A}_2)$ such that $\mathcal{A}_1(1^\lambda)$ ouputs a set $S \subseteq \{0,1\}^{n(\lambda)}$ and state σ, consider an experiment where $K \leftarrow \mathsf{Gen}_F(1^\lambda)$ and $K_S = \mathsf{Punc}_F(K,S)$. Then we have

$$\left|\mathsf{Pr}[\mathcal{A}_2(\sigma, K_S, S, \mathsf{Eval}_F(K,S)) = 1] - \mathsf{Pr}[\mathcal{A}_2(\sigma, K_S, S, U_{m(\lambda)\cdot|S|}) = 1]\right| = \mathsf{negl}(\lambda)$$

  where $\mathsf{Eval}_F(K,S)$ denotes the the concatenation of $(\mathsf{Eval}_F(K,x_1), \ldots, \mathsf{Eval}_F(K,x_k))$, where $S = \{x_1, \ldots, x_k\}$ is the enumeration of the elements of S in lexicographic order and U_ℓ denotes the uniform distribution over ℓ bits.

The GGM tree-based construction of PRFs [37] from one-way function are easily seen to yield puncturable PRFS, as shown in [12, 14, 44]. Thus we have:

Theorem 6. *If one-way functions exist, then for all efficiently computable functions $n(\lambda)$ and $m(\lambda)$, there exists a puncturable PRF family that maps $n(\lambda)$ bits to $m(\lambda)$ bits.*

3.2 Indistinguishability Obfuscation

The definition below is from [27].

Definition 2. A uniform ppt machine $i\mathcal{O}$ is called an Indistinguishability obfuscator for a circuit class $\{C_\lambda\}$, if the following conditions are satisfied:

- For all security parameter λ, all circuit $C \in C_\lambda$, all input x, we have that

$$\mathsf{Pr}[C'(x) = C(x) : C' \leftarrow i\mathcal{O}(\lambda, C)] = 1$$

- For all (not necessarily uniform) ppt adversaries $(\mathcal{A}_0, \mathcal{A}_1)$, there exists a negligible function α, such that the following holds: if $\mathsf{Pr}[\forall x, C_0(x) = C_1(x) : (C_0, C_1, \sigma) \leftarrow \mathcal{A}_0(1^\lambda)] > 1 - \alpha(\lambda)$, then we have

$$|\mathsf{Pr}[\mathcal{A}_1(\sigma, i\mathcal{O}(\lambda, C_0)) = 1] - \mathsf{Pr}[\mathcal{A}_1(\sigma, i\mathcal{O}(\lambda, C_1)) = 1]| \leq \alpha(\lambda)$$

Theorem 7 ([48, 49]). *For every large enough security parameter λ, assuming $2^n\epsilon$-secure functional encryption, there exists an ϵ-secure indistinguishability obfuscator for circuits with input length n.*

In particular, when $n = \log(\lambda)$ and ϵ is negligible in security parameter, iO for n-length circuits, can be based on polynomially secure compact functional encryption.

3.3 Selective-Database Laconic Oblivious Transfer

The definition of laconic oblivious transfer is proposed in [24, 34]. The security notion we need about laconic oblivious transfer is based on work [45].

A laconic oblivious transfer scheme LacOT consists of four algorithms ($\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{Receive}$) with details as follows:

- $\mathsf{crsGen}(1^\lambda)$ takes as input security parameter λ and outputs a common reference string crs.

- $\mathsf{Hash}(\mathsf{crs}, D)$ is a deterministic algorithm that takes as input the crs as well as a database $D \in \{0,1\}^*$, and outputs a hash value h and a state $\widehat{D}$.

- $\mathsf{Send}(\mathsf{crs}, h, L, m_0, m_1)$ takes as input the crs, hash value h, a pair of messages (m_0, m_1) and an index $L \in \mathbb{N}$. It outputs a ciphertext c.

- $\mathsf{Receive}^{\widehat{D}}(\mathsf{crs}, c, L)$ is an algorithm with random access to a database $\widehat{D}$ that takes as input the crs, a ciphertext c and an index $L \in \mathbb{N}$. It outputs a message m.

The scheme LacOT satisfies the following correctness and security properties:

Correctness. We say the scheme LacOT is correct, if for all $D \in \{0,1\}^*$ of size $N = \mathsf{poly}(\lambda)$, all $i \in [N]$ and all $(m_0, m_1) \in \{0,1\}^{p(\lambda)}$, it holds that

$$\mathsf{Pr}\left[\mathsf{Receive}^{\widehat{D}}(\mathsf{crs}, c, L) = m_{D[L]}\right] = 1$$

where $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$, $(h, \widehat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$ and $c \leftarrow \mathsf{Send}(\mathsf{crs}, h, L, m_0, m_1)$.

Selective-database adaptive-message sender privacy against semi-honest receivers. There exists a ppt simulator Sim that satisfies the following:

$$|\mathsf{Pr}[\mathsf{Expt}^{\mathsf{sel}}_{\mathsf{real}}(1^\lambda) = 1] - \mathsf{Pr}[\mathsf{Expt}^{\mathsf{sel}}_{\mathsf{sim}}(1^\lambda) = 1]| \leq \mathsf{negl}(\lambda)$$

where the experiments $\mathsf{Expt}^{\mathsf{sel}}_{\mathsf{real}}(1^\lambda)$ and $\mathsf{Expt}^{\mathsf{sel}}_{\mathsf{sim}}(1^\lambda)$ are in Figure 2:

1. $(D, \mathsf{st}) \leftarrow \mathcal{A}(1^\lambda)$	1. $(D, \mathsf{st}) \leftarrow \mathcal{A}(1^\lambda)$
2. $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$	2. $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$
3. $(h, \widehat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$	3. $(L, m_0, m_1, \mathsf{st}') \leftarrow \mathcal{A}(\mathsf{st}, \mathsf{crs})$
4. $(L, m_0, m_1, \mathsf{st}') \leftarrow \mathcal{A}(\mathsf{st}, \mathsf{crs})$	4. $e \leftarrow \mathsf{Sim}(\mathsf{crs}, D, L, m_{D[L]})$
5. $e \leftarrow \mathsf{Send}(\mathsf{crs}, h, L, m_0, m_1)$	5. $b' \leftarrow \mathcal{A}(\mathsf{crs}, e, \mathsf{st}')$
6. $b' \leftarrow \mathcal{A}(\mathsf{crs}, e, \mathsf{st}')$	
(a) $\mathsf{Expt}^{\mathsf{sel}}_{\mathsf{real}}(1^\lambda)$	(b) $\mathsf{Expt}^{\mathsf{sel}}_{\mathsf{sim}}(1^\lambda)$

Figure 2: Experiments associated with sender privacy for reads

where $|D| = N = \mathsf{poly}(\lambda)$, $L \in [N]$ and $m_0, m_1 \in \{0,1\}^{p(\lambda)}$.

Efficiency. We require that $|h|$ is bounded by a fixed polynomial in λ, and being independent of $|D|$. The runtime of algorithm Hash is $|D| \cdot \mathsf{poly}(\log |D|, \lambda)$, and the runtime of Send and $\mathsf{Receive}$ are $\mathsf{poly}(\log |D|, \lambda)$.

A variant of laconic OT that supports write operation is called updatable laconic OT, defined in the following:

Definition 3 (Updatable laconic OT [24])**.** A laconic OT scheme LacOT is called updatable if it supports the following two algorithms:

- $e_w \leftarrow \mathsf{SendWrite}\left(\mathsf{crs}, h, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|h|}\right)$: On input the common reference string crs, a hash value h, a location $L \in [N]$, bit $b \in \{0,1\}$ and $|h|$ pairs of messages $\{m_{j,0}, m_{j,1}\}_{j=1}^{|h|}$, it outputs a ciphertext e_w.
- $\{m_j\}_{j=1}^{|h|} \leftarrow \mathsf{ReceiveWrite}^{\widehat{D}}(\mathsf{crs}, L, b, e_w)$: On input the common reference string crs, location L, a bit $b \in \{0,1\}$, a ciphertext e_w and random access to state $\widehat{D}$, it updates the state $\widehat{D}$ (such that $D[L] = b$) and outputs messages $\{m_j\}_{j=1}^{|h|}$.

We require an updatable laconic oblivious transfer to additionally satisfy the following properties:

- **Correctness of Writes**: Let database D be of size at most $N = \mathsf{poly}(\lambda)$. Let D^* be a database that is identical to D except that $D^*[L] = b$ for bit $b \in \{0,1\}$. For any sequence of messages $\{m_{j,0}, m_{j,1}\}_{j\in[\lambda]} \in \{0,1\}^{p(\lambda)}$, it holds that

  $$\mathsf{Pr}[m'_j = m_{j,d^*_j}, \forall j \in [|h|] : \{m'_j\}_{j=1}^{|h|} \leftarrow \mathsf{ReceiveWrite}^{\widehat{D}}(\mathsf{crs}, L, b, e_w)] = 1$$

  where $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$, $(d, \widehat{D}) \leftarrow \mathsf{Hash}(\mathsf{crs}, D)$, $(d^*, \widehat{D}^*) \leftarrow \mathsf{Hash}(\mathsf{crs}, D^*)$, and we have

  $$e_w \leftarrow \mathsf{SendWrite}\left(\mathsf{crs}, h, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|h|}\right)$$

- **Selective-database adaptive-message sender privacy against semi-honest receivers with regard to writes**: There exists a ppt simulator $\mathsf{SimWrite}$ satisfies the following

  $$\left|\mathsf{Pr}[\mathsf{Expt}^{\mathsf{wrt}}_{\mathsf{real}}(1^\lambda) = 1] - \mathsf{Pr}[\mathsf{Expt}^{\mathsf{wrt}}_{\mathsf{ideal}}(1^\lambda) = 1]\right| = \mathsf{negl}(\lambda)$$

where experiments $\mathsf{Expt}^{\mathsf{wrt}}_{\mathsf{real}}$ and $\mathsf{Expt}^{\mathsf{wrt}}_{\mathsf{ideal}}$ are defined in Figure 3, where D^* is identical to D except $D^*[L] = b$.

- **Efficiency.** We require that the runtime of algorithms $\mathsf{SendWrite}$ and $\mathsf{ReceiveWrite}$ are $\mathsf{poly}(\log |D|, \lambda)$.

1. $(D, \mathsf{st}) \leftarrow \mathcal{A}(1^\lambda)$
2. $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$.
3. $h = \mathsf{Hash}(\mathsf{crs}, D)$
4. $(L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|h|}, \mathsf{st}) \leftarrow \mathcal{A}(\mathsf{st}, \mathsf{crs})$
5.
6. $e \leftarrow \mathsf{SendWrite}(\mathsf{crs}, h, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|h|})$
7. $b' \leftarrow \mathcal{A}(\mathsf{crs}, e, \mathsf{st}')$.

(a) $\mathsf{Expt}^{\mathsf{wrt}}_{\mathsf{real}}(1^\lambda)$

1. $(D, \mathsf{st}) \leftarrow \mathcal{A}(1^\lambda)$
2. $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$.
3. $h = \mathsf{Hash}(\mathsf{crs}, D)$
4. $(L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|h|}, \mathsf{st}) \leftarrow \mathcal{A}(\mathsf{st}, \mathsf{crs})$
5. $(h^*, \widehat{D}^*) \leftarrow \mathsf{Hash}(\mathsf{crs}, D^*)$
6. $e \leftarrow \mathsf{Sim}(\mathsf{crs}, D, L, b, \{m_{j,h_j^*}\}_{j \in [|h|]})$
7. $b' \leftarrow \mathcal{A}(\mathsf{crs}, e, \mathsf{st}')$.

(b) $\mathsf{Expt}^{\mathsf{wrt}}_{\mathsf{ideal}}(1^\lambda)$

Figure 3: Experiments associated with sender privacy for writes

In [45], the authors show that selective-database laconic OT can be constructed from weakly-selectively secure, single-key public-key functional encryption for circuits, i.e.

Theorem 8 ([45]). *Assuming the existence of public-key functional encryption for circuits, there exists selective-database laconic OT.*

Theorem 9 ([17, 24]). *Assuming the existence of laconic OT, there exists public-key encryption.*

4 Functional Encryption for RAMs

We define a public-key functional encryption scheme for RAM programs [36]. A public-key FE for RAM programs consists of the probabilistic polynomial time (ppt) algorithms $\Pi = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec})$, defined as follows:

- **Setup algorithm.** $\mathsf{Setup}(1^\lambda, T)$: On input security parameter λ, an upper bound T on the running time of the RAM program, the setup algorithm outputs the master secret key MSK and public key pk.
- **Encryption algorithm.** $\mathsf{Enc}(\mathsf{pk}, D)$: On input public key pk and database D, the encryption algorithm outputs the ciphertext CT.
- **Key generation algorithm.** $\mathsf{KeyGen}(\mathsf{MSK}, P)$: On input master secret key MSK, RAM program P, the key generation algorithm outputs the functional key sk_P.
- **Decryption algorithm.** $\mathsf{Dec}^{\mathsf{CT}}(\mathsf{sk}_P)$: On input a functional key sk_P and with random access to ciphertext CT, the decryption algorithm (modeled as a RAM program) outputs the result y.

Definition 4 (Correctness)**.** A public-key functional encryption for RAMs scheme Π is correct, if there exists a negligible $\mathsf{negl}(\cdot)$ such that for any security parameter λ, any database D, for any RAM program P, it holds that

$$\mathsf{Pr}\left[\mathsf{Dec}^{\mathsf{CT}}(\mathsf{sk}_P) = P^D\right] = 1 - \mathsf{negl}(\lambda)$$

where $(\mathsf{pk}, \mathsf{MSK}) \leftarrow \mathsf{Setup}(1^\lambda, T), \mathsf{CT} \leftarrow \mathsf{Enc}(\mathsf{pk}, D)$, $\mathsf{sk}_P \leftarrow \mathsf{KeyGen}(\mathsf{MSK}, P)$ and the probability is taken over the internal randomness of algorithms Setup, Enc and KeyGen.

Succinctness. Unlike the traditional functional encryption for circuits scheme, where the parameters can grow with the worst case runtime of the computation, we require the parameters in the functional encryption for RAMs schemes to have the following efficiency guarantees.

Definition 5 (Succinctness)**.** A public-key functional encryption for RAMs scheme $(\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec})$ satisfies succinctness if the following properties hold:

- $\mathsf{Setup}(1^\lambda, T)$ runs in time $\mathsf{poly}(\lambda, \log(T))$.
- $\mathsf{Enc}(\mathsf{pk}, D)$ runs in time $\mathsf{poly}(\lambda, \log(T), |D|)$.
- $\mathsf{KeyGen}(\mathsf{MSK}, P)$ runs in time $\mathsf{poly}(\lambda, \log(T), |P|)$.
- $\mathsf{Dec}^{\mathsf{CT}}(\mathsf{sk}_P)$ runs in time $\mathsf{poly}(\lambda, T)$.

Remark 3 (Input-Specific Runtime). An astute reader would notice that we only require the decryption time to grow with the worst case time bound, and not with input-specific runtime. Luckily, there is a simple generic transformation that shows how to modify a scheme with worst-case time bound into a scheme that has input-specific runtime: we encourage the reader to refer to [39] for a description of this transformation.

Security. Our security notion is modeled along the same lines as FE for circuits. We only focus on selective security in this work.

Definition 6 (Selective security)**.** A public-key FE for RAMs scheme Π is selectively secure if for any ppt adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that

$$\mathsf{Adv}^{\mathsf{pfe}}_{\Pi,\mathcal{A}}(1^\lambda) = \left|\Pr[\mathsf{Expt}^{\mathsf{pfe}}_{\Pi,\mathcal{A}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Expt}^{\mathsf{pfe}}_{\Pi,\mathcal{A}}(1^\lambda, 1) = 1]\right| \leq \mathsf{negl}(\lambda)$$

for any sufficiently large security parameters λ, where $\mathsf{Expt}^{\mathsf{pfe}}_{\Pi,\mathcal{A}}(1^\lambda, b)$ is defined via the following experiment:

1. **Setup phase**: The challenger computes $(\mathsf{pk}, \mathsf{MSK}) \leftarrow \mathsf{Setup}(1^\lambda, T)$.
2. **Challenge phase**: On input 1^λ, the adversary submits (D_0, D_1), and the challenger replies with pk and $\mathsf{CT} \leftarrow \mathsf{Enc}(\mathsf{pk}, D_b)$.
3. **Query phase**: The adversary adaptively queries the challenger with any RAM program P such that $P^{D_0} = P^{D_1}$. The challenger replies with $\mathsf{sk}_P \leftarrow \mathsf{KeyGen}(\mathsf{MSK}, P)$.
4. **Output phase**: The adversary outputs guess b', which is defined as the output of the experiment.

5 Succinct Reusable Garbled RAM

We first start with the definition of succinct reusable garbled RAM. This will be followed by the construction of succinct UMA-secure reusable GRAM. Finally, we give a transformation from UMA security to full security.

5.1 Syntax and Security Definition

A succinct reusable garbled RAM scheme consists of PPT algorithms $\mathsf{GRAM} = (\mathsf{GrbDB}, \mathsf{GProg}, \mathsf{GEval})$, with details as follows:

- $\mathsf{GrbDB}(1^\lambda, D, T, 1^Q)$: On input security parameter λ, time upper bound T, collusion upper bound Q, a database D, output the garbled database encoding $\widehat{D}$ along with secret key sk.
- $\mathsf{GrbProg}(\mathsf{sk}, P)$: On input secret key sk, and a RAM program P, output the garbled program $\widehat{P}$.
- $\mathsf{GEval}^{\widehat{D}}\left(\widehat{P}\right)$: On input garbled program $\widehat{P}$, database encoding $\widehat{D}$, output y.

Correctness. For correctness, we require that for any program P, any database D, we have that

$$\mathsf{Pr}\left[\mathsf{GEval}^{\widehat{D}}\left(\widehat{P}\right) = P^D()\right] = 1$$

where $(\widehat{D}, \mathsf{sk}) \leftarrow \mathsf{GrbDB}(1^\lambda, T, D)$, and $\widehat{P} \leftarrow \mathsf{GrbProg}(\mathsf{sk}, P)$.

Succinctness. We define succinctness property of garbled RAM. In the definition below, we note the dependence of $\log T$ is implicit since $\log T$ is at most the security parameter.

Definition 7 (Weak succinctness)**.** A garbled RAM scheme $\mathsf{GRAM} = (\mathsf{GrbDB}, \mathsf{GrbProg}, \mathsf{GEval})$ satisfies the weak succinctness property if the following holds:

- $\mathsf{GrbDB}(1^\lambda, T, 1^Q, D)$ runs in time $\mathsf{poly}\,(\lambda, \log T, Q, |D|)$.
- $\mathsf{GrbProg}(\mathsf{sk}, P)$ runs in time $\mathsf{poly}(\lambda, T, \log Q, \log |D|, |P|)$.
- $\mathsf{GEval}^{\widehat{D}}\left(\widehat{P}\right)$ runs in time $\mathsf{poly}(\lambda, t, |P|, \log Q, \log |D|)$.

Definition 8 (Succinctness)**.** A garbled RAM scheme $\mathsf{GRAM} = (\mathsf{GrbDB}, \mathsf{GrbProg}, \mathsf{GEval})$ satisfies (full) succinctness property if the following holds:

- It satisfies the weak succinctness;
- $\mathsf{GrbProg}(\mathsf{sk}, P)$ runs in time $\mathsf{poly}(\lambda, \log T, \log Q, \log |D|, |P|)$, instead of T.

Reusable Security. We define a notion of reusable security that will be compatible with the security definition of FE for RAMs.

To define reusable security, we first describe the experiment below.

$\underline{\mathsf{Expt}^{\mathcal{A}}(1^\lambda, b)}$:

- $\mathcal{A}$ submits two databases D_0 and D_1, a collusion bound Q (or $\perp$ for unbounded GRAM scheme), and a running time bound encoded in unary 1^T.
- The challenger responds back with database encoding $\widehat{D_b}$.
- Proceeding adaptively, $\mathcal{A}$ submits RAM programs P_0, P_1. The challenger checks that $P_0^{D_0}() = P_1^{D_1}()$ and each program executes for the same number of time steps. It also checks that $|D_0| = |D_1|$. If both the checks fail, it aborts; otherwise, it sends the garbled program $\widehat{P_b}$ and garbled input $\widehat{x_b}$. $\mathcal{A}$ repeats this step for $Q = \mathsf{poly}(\lambda)$ times.
- $\mathcal{A}$ outputs b'. The output of the experiment is b'.

Definition 9 ((Indistinguishability) reusability)**.** A garbled RAM scheme $(\mathsf{GrbDB}, \mathsf{GrbProg}, \mathsf{Eval})$ satisfies (indistinguishability) reusability property if the following holds for every ppt adversary $\mathcal{A}$:

$$\left|\mathsf{Pr}[0 \leftarrow \mathsf{Expt}^{\mathcal{A}}(1^\lambda, 0)] - \mathsf{Pr}[0 \leftarrow \mathsf{Expt}^{\mathcal{A}}(1^\lambda, 1)]\right| \leq \mathsf{negl}(\lambda)$$

Remark 4. Our construction actually satisfies a stronger security of simulation security, where simulated version of GrbDB only takes as input $(1^\lambda, 1^{|D|})$, and the simulated version of $\mathsf{GrbProg}$ only takes as input $(\mathsf{sk}, 1^{|P|}, y)$. Note that for this definition, simulation security is in fact equivalent to indistinguishability security[¶¶].

Unbounded Reusability. Ideally, we would like the garbled database encoding to be reusable by a priori unbounded number of garbled programs. We capture this in the formal definition below.

Definition 10 (Unbounded reusability)**.** In addition to succinctness, a succinct garbled RAM scheme satisfies unbounded reusability, if the algorithm GrbDB takes $Q = \perp$ and all algorithms run in time independent of Q, for example, GrbDB runs in time $\mathsf{poly}(\lambda, \log T, |D|)$.

5.2 Succinct UMA Reusable GRAM

To construct succinct reusable GRAM, we start by constructing a succinct garbled RAM scheme that only satisfies a weaker notion of reusable security, which we call UMA security.

UMA security. UMA security is defined similar as the indistinguishability security above, except that the challenger in addition to checking $P_0^{D_0}() = P_1^{D_1}()$, she also checks that $D_0 = D_1$, and every step circuits in $P_0^{D_0}(), P_1^{D_1}()$ at the same time step output the exact same output.

Ingredients. We use the following ingredients in our construction:

- Selective-database updatable laconic oblivious transfer ($\mathsf{crsGen}, \mathsf{Hash}, \mathsf{Send}, \mathsf{SendWrite}, \mathsf{Receive}, \mathsf{ReceiveWrite}$).
- A puncturable PRF ($\mathsf{PRF.Gen}, \mathsf{PRF.Eval}, \mathsf{PRF.Punc}$).
- Indistinguishability obfuscation $\mathrm{i}\mathcal{O}$ for circuits with log-sized inputs.

Construction. We construct $\Pi = (\mathsf{GrbDB}, \mathsf{GrbProg}, \mathsf{GEval})$ as follows:

- $\mathsf{GrbDB}(1^\lambda, D, 1^Q, T_{\max})$: On input security parameter λ, database D and running time upper bound $T_{\max}$, it does the following:
  1. Sample $\mathsf{crs} \leftarrow \mathsf{crsGen}(1^\lambda)$ and compute $(d, \widehat{D}) = \mathsf{Hash}(\mathsf{crs}, D)$
  2. Output $\widehat{D}$ as garbled database and $(d, \mathsf{crs}, Q, T_{\max})$ as the secret key sk.
- $\mathsf{GrbProg}(\mathsf{sk}, P)$: On input secret key sk and program P, it does:
  1. Sample a PRF key $K \leftarrow \mathsf{PRF.Gen}(1^\lambda)$.
  2. For each step $\tau \in [2, T], k \in [\lambda+n+1]$ and $b \in \{0,1\}$, let $\mathsf{lab}_{k,b}^\tau = \mathsf{PRF}_K(\tau||k||b)$.
  3. We use $\{\mathsf{lab}_{k,b}^\tau\}$ to denote $\{\mathsf{lab}_{k,b}^\tau\}_{k\in[\lambda+n+1], b\in\{0,1\}}$.
  4. Output $\widehat{P} = (\mathrm{i}\mathcal{O}(PG[P, \mathsf{crs}, K, d]), \{\mathsf{lab}_{k,d_k}^1\}_{k\in[\lambda]}, \{\mathsf{lab}_{k+\lambda,0}^1\}_{k\in[n+1]})$, where PG is described in Figure 4.
     Note: we pad the circuit PG such that its size is $|P| \cdot \mathsf{poly}(\lambda, \log|D|, \log T)$ bits. This will become clear later in the security proof.

[¶¶] In general these two notions are not equivalent: in our setting, they are equivalent since we only consider programs with boolean outputs.

Program generator circuit PG

Hardwired values: the program P, the CRS string crs, the PRF key K, the initial digest d

Input: step number τ

(a) Compute $C_{\mathsf{CPU}} \leftarrow P(\tau)$. That is, P on input time step τ outputs the step circuit C_{CPU}.

(b) Output $\mathsf{GarbleCkt}\left(1^\lambda, C\left[C_{\mathsf{CPU}}, \mathsf{crs}, \tau, \{\mathsf{lab}_{k,b}^{\tau+1}\}\right], \{\mathsf{lab}_{k,b}^{\tau}\}; \mathsf{PRF}_K(\tau)\right)$, where the circuit C is described in Figure 5.

Figure 4: Description of program generator circuit PG

Step circuit C

Hardwired values: A step circuit C_{CPU}, the CRS string crs, the step number τ and a set of labels $\{\mathsf{lab}_{k,b}\}$

Input: A digest d, state st and read value rv.

(a) Compute $(\mathsf{st}', \mathsf{op}, \mathsf{addr}, \mathsf{wb}) = C_{\mathsf{CPU}}(\mathsf{st}, \mathsf{rv})$.

(b) If st' is in abort state, reset $\mathsf{lab}_{k,b} = b$ for $k \in [\lambda+1, \lambda+n]$ and $b \in \{0,1\}$.

(c) If $\mathsf{op} = \mathsf{write}$, compute $e_w \leftarrow \mathsf{SendWrite}\left(\mathsf{crs}, d, \mathsf{addr}, \mathsf{wb}, \{\mathsf{lab}_{k,b}\}_{k\in[\lambda],b\in\{0,1\}}\right)$. Output

$$\left(\mathsf{op}, \mathsf{addr}, e_w, \mathsf{wb}, \{\mathsf{lab}_{k,\mathsf{st}'[k-\lambda]}\}_{i\in[n],k\in[\lambda+1,\lambda+n]}, \mathsf{lab}_{n+\lambda+1,0}\right)$$

(d) Otherwise, compute $e \leftarrow \mathsf{Send}\left(\mathsf{crs}, d, \mathsf{addr}, \{\mathsf{lab}_{\lambda+n+1,b}\}_{b\in\{0,1\}}\right)$. Output

$$\left(\mathsf{op}, \mathsf{addr}, \{\mathsf{lab}_{k,d_k}\}_{k\in[\lambda]}, \{\mathsf{lab}_{k,\mathsf{st}'[k-\lambda]}\}_{k\in[\lambda+1,\lambda+n]}, e\right)$$

Figure 5: Description of step circuit C

- $\mathsf{GEval}^{\widehat{D}}\left(\widehat{P}\right)$: With random access to $\widehat{D}$ and on input garbled program $\widehat{P}$,
  1. Extract $\widetilde{\mathsf{lab}} \leftarrow \{\mathsf{lab}_{k,x_k}^1\}_{k\in[\lambda+n+1]}$ from the garbled program
  2. For τ from 1 to T,
     - Invoke the i$\mathcal{O}$ program on τ to obtain $\widehat{C}_\tau$.
     - Compute $\left(\mathsf{op}, \mathsf{addr}, A, \{\mathsf{lab}_k\}_{k\in[\lambda+1,\lambda+n]}, B\right) = \mathsf{EvalCkt}\left(\widehat{C}_\tau, \widetilde{\mathsf{lab}}\right)$.
     - If the labels corresponding to st are in plain-text, abort the loop
     - If $\mathsf{op} = \mathsf{write}$, parse A as (e_w, wb) and B as $\{\mathsf{lab}_k\}_{k\in[\lambda+1,\lambda+n]}$. Compute $\{\mathsf{lab}_k\}_{k\in[\lambda]} \leftarrow \mathsf{ReceiveWrite}^{\widehat{D}}(\mathsf{crs}, \mathsf{addr}, \mathsf{wb}, e_w)$.
     - Otherwise, parse A as $\{\mathsf{lab}_k\}_{k\in[\lambda+n]}$ and B as e. Compute $\mathsf{lab}_{\lambda+n+1} \leftarrow \mathsf{Receive}^{\widehat{D}}(\mathsf{crs}, \mathsf{addr}, e)$.
     - Let $\widetilde{\mathsf{lab}} \leftarrow \{\mathsf{lab}_{k,x_k}\}_{k\in[\lambda+n+N]}$
  3. Output $\{\mathsf{lab}_k\}_{k\in[\lambda+1,\lambda+n]}$.

Correctness. We can prove the correctnss of our construction using an inductive argument that for each step τ, the state st and databases are updated correctly at the end of execution of step circuit. The base case is $\tau = 0$. For $\tau \neq 0$, observe that if $\mathsf{op} = \mathsf{write}$, then algorithm Eval updates the database D_j and its associated digest, where D_j is the corresponding database for write location addr. Otherwise, if $\mathsf{op} = \mathsf{read}$, the labels recovered in Eval step 2 correspond to the value in the location addr as requested.

Succinctness.

1. By the efficiency of laconic OT, GrbDB runs in time $\mathsf{poly}(\lambda, |D|) + \log Q + \log T_{\max}$.
2. By the efficiency of indistinguishability obfuscation, $\mathsf{GrbProg}$ runs in time $\mathsf{poly}(\lambda, \log T, \log |D|, |P|)$.
3. Finally, GEval runs in time $t \cdot \mathsf{poly}(\lambda, \log T, \log |D|, |P|)$, as it will abort execution once the new state is in abort state.

We now prove that the above scheme is secure.

Theorem 10. *Assuming the security of selective-database updatable laconic oblivious transfer, puncturable PRF and i$\mathcal{O}$ with log-sized inputs, there exists a succinct (unbounded) reusable garbled RAM scheme satisfying UMA security.*

The crux of the proof is to show that the above construction satisfies reusable security. Consider a PPT adversary $\mathcal{A}$. Let $\mathcal{A}$ submit Q program pairs $(P_{1,0}, P_{1,1}), \ldots, (P_{Q,0}, P_{Q,1})$. We employ a standard hybrid argument.

$\underline{\mathsf{Hyb}_k^{prog}}$: In this hybrid, the challenger generates the database encoding $\widehat{D}$ honestly. For $i \leq k-1$, it generates the garbled program $\widehat{P_{i,0}}$ and for $i \geq k$, it generates the garbled program to be $\widehat{P_{i,1}}$.

If we show that $\mathsf{Hyb}_k^{prog} \approx_c \mathsf{Hyb}_{k+1}^{prog}$, for any $k \in \{1, \ldots, Q-1\}$ then this implies that $\mathsf{Hyb}_0^{prog} \approx_c \mathsf{Hyb}_{Q+1}^{prog}$; thus proving that the scheme satisfies reusability security. Due to the space limit, we only describe a sketch here. The full proof is presented in the full version.

Instantiation. Combining the above theorem with the FE-based iO construction [48, 49] and FE-based laconic OT construction [45], we arrive at the following corollary.

Corollary 11. *Assuming the existence of public-key functional encryption for circuits, there exists a succinct (unbounded) garbled RAM scheme satisfying UMA security.*

Bounded-key setting. For the bounded-key setting, since we only aim for the weak succinctness, we can consider the same construction as before except that we can instantiate i$\mathcal{O}$ with an *inefficient* i$\mathcal{O}$ scheme, i.e., a scheme that outputs the truth table of the circuit being obfuscated. Note that since we only consider i$\mathcal{O}$ for logarithmic inputs, the size of the truth table is still polynomial in λ. As a result, the running time of $\mathsf{GrbProg}$ is now $T \cdot \mathsf{poly}(\lambda, \log T, \log |D|, |P|)$. Thus, we have the following theorem.

Theorem 12. *Assuming the existence of selective-database updatable laconic oblivious transfer, there exists a weakly-succinct (unbounded) garbled RAM scheme satisfying UMA security.*

5.3 Succinct Reusable GRAM: From UMA to Full Security

In this section, we will present the construction of (fully) succinct reusable garbled RAM. We present a transformation that converts a succinct reusable garbled RAM with UMA security into a succinct reusable garbled RAM scheme with full security. While such UMA to full security setting have been known in the past, they have not been studied in the (parallel) reusable setting, which is the focus of our work.

One of the main ingredients in our construction is an initial-state rewindable ORAM scheme satisfying strong localized randomness property. We start by presenting a construction of this.

5.3.1 Rewindable ORAM with Strong Localized Randomness

Alternate Formulation of ORAMs. Before we recall the definition of strong localized randomness, we first consider an alternate (equivalent) definition of ORAM schemes. We consider a pair of PPT algorithms $(\mathsf{OData}, \mathsf{OProg})$.

Algorithm $\mathsf{OData}(1^\lambda, D)$ takes as input security parameter λ, database $D \in \{0,1\}^N$ and outputs the oblivious database D^* and some client key ck. Algorithm $\mathsf{OProg}(1^\lambda, 1^{\log N}, 1^T, P, \mathsf{ck})$ takes as input security parameter λ, memory size N, runtime T, a RAM program P, and the client key ck, and outputs a compiled program P^*, which is a RAM program that instead operates on D^*.

Strong Localized Randomness. The additional property we need from ORAM is called strong localized property from an ORAM scheme. The definition we use here is based on [31] and is stronger than the original definition.

Let $D \in \{0,1\}^N$ be any database and (P, x) be any program/input pair. Let the step circuits of P^* be indicated by $\{C^\tau_{CPU}\}_{\tau \in [T']}$ and R be the contents of the random tape used in the execution.

Definition 11 (Strong localized randomness)**.** We say that an ORAM scheme has strong localized randomness property if for any sequence of memory accesses of length T, there exists a sequence of efficiently computable values $1 = \tau_1 < \tau_2 < \cdots < \tau_m = T' + 1$, where $\tau_t - \tau_{t-1} \leq \mathsf{poly}(\log N)$ for all $t \in [2, m]$, such that

1. For every $j \in [m-1]$, there exists an interval I_j of size $\mathsf{poly}(\log N, \lambda)$, such that for any $\tau \in [\tau_j, \tau_{j+1}]$, the random tape accessed by C^τ_{CPU} is given by R_{I_j}.
2. For every $j, j' \in [m-1]$ and $j \neq j'$, it holds that $I_j \cap I_{j'} = \emptyset$.
3. There exists a PPT procedure CkSim that takes as input $(\tau_k, \tau_{k+1}, \mathsf{ck})$ and outputs ck'. It has the following guarantee: there exists a PPT algorithm that takes as input τ_i for $i \neq k$, ck', R_{I_i} and outputs the correct (real world) memory access pattern.

   Furthermore, the following security guarantee is satisfied. $\forall j \in [m]$, $\exists k < j$, the following distributions are computationally indistinguishable:

   - $R_{\setminus I_k \cup I_j}$ (where $R_{\setminus I_k \cup I_j}$ denotes the content of random tape except in positions $I_k \cup I_j$), $\mathsf{ck}' := \mathsf{CkSim}(\tau_k, \tau_{k+1}, \mathsf{ck})$, the memory accesses for $\tau \in [\tau_k, \tau_{k+1})$[***] and the memory accesses for $\tau \in [\tau_j, \tau_{j+1})$.
   - $R_{\setminus I_k \cup I_j}$, $\mathsf{ck}' := \mathsf{CkSim}(\tau_k, \tau_{k+1}, \mathsf{ck})$ and the memory accesses for $\tau \in [\tau_k, \tau_{k+1})$ and uniformly random memory accesses (with the same length as the memory accesses for $\tau \in [\tau_j, \tau_{j+1})$).

Theorem 13 (ORAM with strong localized randomness [31])**.** *Assuming one-way functions, there exists ORAM with strong localized randomness property.*

[***] $[\tau_k, \tau_{k+1})$ denotes the contents of the random tape starting from τ_k^{th} position to $(\tau_{k+1}-1)^{th}$ position.

We remark that even though the definition of strong localized randomness in [31] does not talk about CkSim, they implicitly constructed such a simulator at the end of Appendix B, and their proof in Appendix D.1 implicitly relied on the fact that such simulation is possible.

Our Construction. We present our construction of ISR-ORAM with strong localized randomness property.

Theorem 14. *Assuming the existence of ORAM with strong localized randomness and (unbounded) PK-DEPIR, there exists unbounded ISR-ORAM with strong localized randomness.*

Proof. The proof is done via two steps. First, we construct an ORAM with initially-empty database and strong localized randomness property, from an ORAM with strong localized randomness property; next, we add the ISR property to the construction via using PK-DEPIR.

From Large Initial DB to Empty Initial DB. To prove the theorem, first we build an ORAM with initially-empty database *and* strong localized randomness from ORAM with only strong localized randomness property. The requirements for ORAM with an initially-empty database are essentially the same as ordinary ORAM, except that we restrict the scheme to having an empty database at the beginning and allow the size of the database to grow as the number of operations increase. (On the other hand, traditional ORAM works on a fixed-size database who is given in its entirety at the beginning.) Furthermore, it needs to be able to achieve this without knowing an upper bound on the number of operations a priori.

The construction is as follows:

1. Initialize an ORAM D of length C; (at the beginning take C to be any constant, say 1)
2. Read/write to the ORAM until ORAM program has performed over C writes;
3. Reinitialize another ORAM D' of length $2C$ and copy data from D to D';
4. Discard D and take D' to be the new D, return to 2.

Despite possibly running in time linear in the size of the entire database for a single write, this construction will only have amortized cost constant times the original read/write amortized cost. This is because every time we are expanding the database from size S to $2S$, while this costs $O(S)$ operations, it means that we have performed $S/2$ operations since the last expansion. Therefore, we can average the cost of this expansion into each operation, and thus on average the cost for each operation is independent of S. On the other hand, strong localized randomness property follows naturally as we are using an ORAM with strong localized randomness as our building block. Finally, since by construction the expansion only depends on the running time/the number of writes, the security properties are preserved.

Generically Achieving Initial-State Rewindable Property. Next, we recall the construction of ISR-ORAM. The idea is that we will have a read-only ORAM instantiated by PK-DEPIR and another read-write (initially-empty) ORAM "cache" instantiated by the actual ORAM. The overall client state will consists of (ck, k), where ck is the client state for the initially-empty ORAM, and k is the (public) key for the PK-DEPIR. Whenever we do a read, we read from both databases and return the cached result if cache read results in a hit. For writes, we simply write directly to the cache.

To construct unbounded ISR-ORAM with SLR, we simply change the construction above to use the initially-empty ORAM with SLR instead of initially-empty ORAM. Note that the construction has the efficiency we desire as argued above.

We now argue that it satisfies the strong localized randomness property. The first two properties follow naturally, as there are only two places where we use randomness; for the ORAM, this follows as we are using an ORAM with strong localized randomness property; for the DEPIR, this follows as the randomness used by DEPIR is freshly sampled for every access and therefore independent of everything else. To argue the third property, CkSim simulates ck by calling the underlying CkSim of ORAM with SLR, and output the public key k for the PK-DEPIR as is. Using SLR of the initially-empty ORAM, the memory access pattern for ISR-ORAM is indistinguishable from random; and by the security of PK-DEPIR (where the distinguisher gets access to the key), the memory access pattern for PK-DEPIR is indistinguishable from random.

Finally, it is apparent that for this construction, if we start with ORAM without SLR instead of ORAM with SLR, and PK-DEPIR instead of B-bounded SK-DEPIR, we will end up with B-bounded ISR-ORAM without SLR property by the same argument. □

We are now ready to present the construction of succinct reusable GRAM in the full security setting.

Ingredients. We use the following cryptographic tools:

- Unbounded ISR-ORAM scheme $(\mathsf{OData}, \mathsf{OProg})$ with strong localized randomness (Section 5.3.1).
- UMA-secure reusable garbled RAM scheme (Section 5.2).
- Puncturable PRF [12, 14, 44] $(\mathsf{PRF.Gen}, \mathsf{PRF.Eval}, \mathsf{PRF.Punc})$.
- Timed encryption scheme [31] $(\mathsf{TE.KeyGen}, \mathsf{TE.Enc}, \mathsf{TE.Dec}, \mathsf{TE.Constrain})$. Let M be the output length of $\mathsf{TE.Enc}$ when encrypting single bit messages.

Construction. We describe the succinct reusable (fully-secure) GRAM $(\mathsf{GrbDB}, \mathsf{GrbProg}, \mathsf{GEval})$ below:

- $\mathsf{GrbDB}(1^\lambda, D, 1^Q, T_{\max})$: On input security parameter λ, database D and running time upper bound $T_{\max}$,
  1. Sample $K \leftarrow \mathsf{TE.KeyGen}(1^\lambda)$.
  2. For $i \in [N]$, compute $D'[i] \leftarrow \mathsf{TE.Enc}(K, 0^\lambda, D[i])$.
  3. Compute $(D^*, \mathsf{ck}) \leftarrow \mathsf{OData}(1^\lambda, D')$.
  4. Run $\mathsf{UGRAM.GrbDB}(1^\lambda, D^*, T'(T_{\max}))$ to obtain $(\mathsf{sk}, \widehat{D})$, where $T'(\cdot)$ is a polynomial corresponding to the running time blow-up of using the ORAM scheme.
  5. Output $\widehat{D}$ as garbled memory and $(\mathsf{sk}, K, \mathsf{ck})$ as secret key SK.
- $\mathsf{GrbProg}(\mathsf{SK}, P)$: On input secret key $\mathsf{SK} = (\mathsf{sk}, K, \mathsf{ck})$ and a program P,
  1. Generate a puncturable PRF key $K' \leftarrow \mathsf{PRF.Gen}(1^\lambda)$.
  2. Compute $P^* \leftarrow \mathsf{OProg}(1^\lambda, N, 1^T, P, \mathsf{ck})$, where P^* runs in time T'.
  3. Construct a RAM program P' such that on input $\tau \in [T']$, do
     (a) Compute $K[\tau] \leftarrow \mathsf{TE.Constrain}(K, \tau)$.
     (b) Let $\tau_1, \ldots, \tau_m$ be the sequence of values guaranteed by the strong localized randomness property of the ORAM scheme.

(c) Let $j \in [m-1]$ such that $\tau \in [\tau_j, \tau_{j+1})$ and $C_{\mathsf{CPU}}^{P^*} \leftarrow P^*(\tau)$. Output $C_{\mathsf{CPU}}^{\tau} = \mathsf{SC}_\tau[C_{\mathsf{CPU}}^{P^*}, \tau, K[\tau], I_j, K']$. The circuit SC is described in figure 6.

Note: We need to pad the program P' such that the total size is $|P| \cdot \mathsf{poly}(\lambda, \log D, \log T)$ bits.

4. Compute and output $\widehat{P} \leftarrow \mathsf{UGRAM.GProg}(\mathsf{sk}, P')$.

Step circuit SC_τ

Hardwired values: A circuit C_{CPU}, step number τ, constrained key $K[\tau]$, interval I_j, and the key K'.
Input: A Ciphertext c_{CPU} and a encrypted data X.

(a) Compute decryption as $\mathsf{rv} = \mathsf{TE.Dec}(K[\tau], X)$ and $\mathsf{st} = \mathsf{TE.Dec}(K[\tau], c_{\mathsf{CPU}})$.
(b) Compute $R_{I_j} = \mathsf{PRF.Eval}(K', I_j)$.
(c) Compute $(\mathsf{st}', \mathsf{op}, \mathsf{addr}', \mathsf{wb}) = C_{\mathsf{CPU}}(\mathsf{st}, \mathsf{rv}; R_{I_j})$.
(d) If $\tau = T'$, then output $c'_{\mathsf{CPU}} = \mathsf{st}'$. Else, compute $c'_{\mathsf{CPU}} \leftarrow \mathsf{TE.Enc}(K[\tau], \mathsf{st}')$.
(e) Else if $\mathsf{op} = \mathsf{write}$, compute $X' \leftarrow \mathsf{TE.Enc}(K[\tau], \tau, \mathsf{wb})$. Output $(c'_{\mathsf{CPU}}, \mathsf{op}, \mathsf{addr}, X')$.
(f) Else if $\mathsf{op} = \mathsf{read}$, output $(c'_{\mathsf{CPU}}, \mathsf{op}, \mathsf{addr}, \perp)$

Figure 6: Description of step circuit $C_{\mathsf{CPU}}^{\tau}[\tau, I_j, K[\tau], K']$

- $\mathsf{GEval}^{\widehat{D}}(\widehat{P})$: With random access to garbled database $\widehat{D}$ and input $\widehat{P}$, it computes and outputs $y = \mathsf{UGRAM.GEval}^{\widehat{D}}(\widehat{P})$.

Theorem 15. *Assuming the existence of public-key functional encryption for circuits and unbounded PK-DEPIR, there exists a succinct reusable garbled RAM scheme.*

Due to the space constraints, we present the proof in the full version.

Bounded Setting. We observe that our techniques can be adapted to get bounded reusable garbled RAM albeit satisfying the weaker succinctness property.

Theorem 16. *Assuming the existence of selective-database updatable laconic oblivious transfer, there exists a weakly-succinct bounded reusable garbled RAM scheme.*

Proof. To put our construction to the Q-bounded-key setting, we implement the following changes for the construction above:

1. UGRAM is replaced by the weakly-succinct reusable UMA GRAM we constructed in Theorem 12;

2. Unbounded ISR-ORAM with strong localized randomness property is replaced with $(Q \cdot T_{\max})$-bounded ISR-ORAM without strong localized randomness property, which can be constructed from one way functions, as we show in Theorem 14.

Even though we lose the strong localized randomness property, since we only need weak succinctness, we can get around the issue by hardwiring all the randomness for the program. Furthermore, as we will only generate at most $Q \cdot T_{\max}$ queries to ISR-ORAM, intuitively, we can simply invoke the security proof above to argue security for the new construction. We present the full proof also in the full version. □

6 Collusion-Resistant Public-Key FE: from Circuits to RAMs

In this part, we show how to construct public-key FE for RAMs from public-key FE for circuits. We use the following tools:

- Public-key FE scheme for circuits scheme $\widetilde{\mathsf{FE}}$.
- Succinct reusable garbled RAM scheme GRAM, where the length of randomness used in algorithm $\mathsf{GRAM.GrbProg}$ is ℓ_1, the length of garbled program is ℓ_2 and the length of garbling key is λ.
- Pseudorandom function $\mathsf{PRF}_1 : \mathcal{K} \times \{0,1\}^\lambda \to \{0,1\}^{\ell_1}$, and $\mathsf{PRF}_2 : \mathcal{K} \times \{0,1\}^\lambda \to \{0,1\}^{\ell_2}$ where $\mathcal{K}$ is the space of keys of size λ.

We construct public-key functional encryption for RAMs scheme $\mathsf{FE} = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec})$ as follows:

- $\mathsf{Setup}(1^\lambda, T)$: On input security parameter λ and upper time bound T,
  1. Compute $(\widetilde{\mathsf{FE}}.\mathsf{MSK}, \widetilde{\mathsf{FE}}.\mathsf{pk}) \leftarrow \widetilde{\mathsf{FE}}.\mathsf{Setup}(1^\lambda)$.
  2. Output $\mathsf{MSK} = \widetilde{\mathsf{FE}}.\mathsf{MSK}, \mathsf{pk} = \widetilde{\mathsf{FE}}.\mathsf{pk}$.
- $\mathsf{Enc}(\mathsf{pk}, D)$: On input public key $\mathsf{pk} = \widetilde{\mathsf{FE}}.\mathsf{pk}$ and database D,
  1. Run the garbling database algorithm,
  $$(\widehat{D}, \mathsf{GRAM.sk}) \leftarrow \mathsf{GRAM.GrbDB}(1^\lambda, D, T)$$
  2. Choose a random PRF key K_1 from PRF key space $\mathcal{K}$.
  3. Compute $\widetilde{\mathsf{FE}}.\mathsf{CT} \leftarrow \widetilde{\mathsf{FE}}.\mathsf{Enc}\left(\mathsf{pk}, (\mathsf{GRAM.sk}, K_1, 0^\lambda, 0)\right)$.
  4. Output ciphertext as $\mathsf{CT} = \left(\widehat{D}, \widetilde{\mathsf{FE}}.\mathsf{CT}\right)$.
- $\mathsf{KeyGen}(\mathsf{MSK}, P)$: On input master secret key $\mathsf{MSK} = (\widetilde{\mathsf{FE}}.\mathsf{MSK}, T)$, a RAM program P,
  1. Sample random string $\tau \leftarrow \{0,1\}^\lambda$, and $r \leftarrow \{0,1\}^{\ell_2}$.
  2. Compute $\widetilde{\mathsf{FE}}.\mathsf{sk}_P \leftarrow \widetilde{\mathsf{FE}}.\mathsf{KeyGen}(\widetilde{\mathsf{FE}}.\mathsf{MSK}, C[P, r, \tau])$ for circuit $C[P, r, \tau]$ as described in Figure 7.
  3. Output $\mathsf{sk}_P = \widetilde{\mathsf{FE}}.\mathsf{sk}_P$.

$$C\left[P, r, \tau\right](\mathsf{GRAM.sk}, K_1, K_2, \beta)$$

Hardwired Values: RAM program P, random strings τ and r.
Input: $(\mathsf{GRAM.sk}, K_1, K_2, \tau, \beta)$.
If $\beta = 1$, then output $r \oplus \mathsf{PRF}_2(K_2, \tau)$.
Else $\beta = 0$,
(a) Run $\mathsf{GRAM.GrbProg}(\mathsf{sk}, P; \mathsf{PRF}_1(K_1, \tau))$ to obtain $\widehat{P}$.
(b) Output garbled program $\widehat{P}$.

Figure 7: Description of circuit $C\left[P, r, \tau\right](\mathsf{GRAM.sk}, K_1, K_2, \beta)$

- $\mathsf{Dec}^{\mathsf{CT}}(\mathsf{sk}_P)$: On input secret key sk_P and random access to ciphertext CT, the decryption algorithm does:
  1. Parse the functional key sk_P as $\widetilde{\mathsf{FE}}.\mathsf{sk}_P$.
  2. Parse the ciphertext CT as $(\widehat{D}, \widetilde{\mathsf{FE}}.\mathsf{CT})$.
  3. Compute $\widehat{P} = \widetilde{\mathsf{FE}}.\mathsf{Dec}\left(\widetilde{\mathsf{FE}}.\mathsf{sk}_P, \widetilde{\mathsf{FE}}.\mathsf{CT}\right)$.
  4. Compute and output $y \leftarrow \mathsf{GRAM.GEval}\left(\widehat{P}, \widehat{D}\right)$.

Correctness. For any RAM program P, database D, let $\mathsf{CT} \leftarrow \mathsf{Enc}(\mathsf{pk}, D)$, and $\mathsf{sk}_P \leftarrow \mathsf{KeyGen}(\mathsf{MSK}, P)$, where $(\mathsf{pk}, \mathsf{MSK})$ are generated as above. Parse CT as $(\widehat{D}, \widetilde{\mathsf{FE}}.\mathsf{CT})$, and $\mathsf{sk}_P = \widetilde{\mathsf{FE}}.\mathsf{sk}_P$. The correctness of $\widetilde{\mathsf{FE}}$ guarantees that $\widehat{P} = \mathsf{GRAM.GrbProg}(\mathsf{GRAM.sk}, P; \mathsf{PRF}(K, \tau))$, where $\widehat{P} = \mathsf{Dec}(\mathsf{sk}_P, \mathsf{CT})$. By the correctness of pseudorandom function PRF and FE scheme $\widetilde{\mathsf{FE}}$, it follows that the output of $\mathsf{GEval}\left(\widehat{P}, \widehat{D}\right) = P^D()$.

Succinctness. We analyze the succinctness property of the construction as follows:

- $\mathsf{Setup}(1^\lambda, T)$ runs in time $\mathsf{poly}(\lambda, \log(T))$: first observe that $\widetilde{\mathsf{FE}}.\mathsf{Setup}(1^\lambda)$ runs in time $\mathsf{poly}(\lambda, \log(s))$, where s denotes the size of supported circuits. Now we determine an upper bound for s. By the succinctness of GRAM, $\mathsf{GrbProg}(\mathsf{sk}, \cdot; \mathsf{PRF}_1(K_1, \tau))$ can be represented by a circuit of size at most $\mathsf{poly}(\lambda, \log(T), |P|)$; thus, $|C| = \mathsf{poly}(\lambda, \log(T), |P|)$. Thus, $s = \mathsf{poly}(\lambda, \log(T), |P|)$.
- $\mathsf{Enc}(\mathsf{pk}, D)$ runs in time $\mathsf{poly}(\lambda, \log(T), |D|)$: we first note that $\widetilde{\mathsf{FE}}.\mathsf{Enc}(\mathsf{pk}, \mathsf{GRAM.sk})$ runs in time $\mathsf{poly}(\lambda, \log(s))$, while $\mathsf{GRAM.GrbDB}(1^\lambda, D, T)$ runs in time $\mathsf{poly}(\lambda, \log(T), |D|)$.
- $\mathsf{KeyGen}(\mathsf{MSK}, P)$ runs in time $\mathsf{poly}(\lambda, \log(T), |P|)$: $\widetilde{\mathsf{FE}}.\mathsf{KeyGen}(\widetilde{\mathsf{FE}}.\mathsf{MSK}, C[P, r, \tau](\mathsf{GRAM.sk}, K_1, K_2, \beta))$ runs in time $\mathsf{poly}(\lambda, s)$ and from the first bullet, $s = \mathsf{poly}(\lambda, \log(T), |P|)$.
- $\mathsf{Dec}^{\mathsf{CT}}(\mathsf{sk}_P)$ runs in time $\mathsf{poly}(\lambda, T)$: the runtime of $\widetilde{\mathsf{FE}}.\mathsf{Dec}(\widetilde{\mathsf{FE}}.\mathsf{sk}_P, \widetilde{\mathsf{FE}}.\mathsf{CT})$ is $\mathsf{poly}(\lambda, \log(T), |P|)$. Moreover, from the succinctness of GRAM, the runtime of $\mathsf{GEval}\left(\widehat{P}, \widehat{D}\right)$ is $\mathsf{poly}(\lambda, t)$, where t is the time taken to execute $P^D()$.

Theorem 17. *If $\widetilde{\mathsf{FE}}$ is a public-key functional encryption for circuits satisfying indistinguishability security, GRAM is a succinct reusable garbled RAM scheme and PRF is a secure pseudorandom function, then the FE for RAMs construction FE described above is selectively secure.*

Proof. We describe the hybrids below; in the first hybrid $\mathsf{Hyb}_{0,b}$, the challenger uses challenge bit $b \xleftarrow{\$} \{0,1\}$ to generate the ciphertexts and in the final hybrids Hyb_4, all the parameters in the system computationally hide b.

$\mathsf{Hyb}_{0,b}$: This correspondes to the real experiment. The challenger computes the following: (i) $(\mathsf{pk}, \mathsf{MSK}) \leftarrow \mathsf{Setup}(1^\lambda, T)$, (ii) $\mathsf{CT}_b \leftarrow \mathsf{Enc}(\mathsf{MSK}, D_b)$, and (iii) $\{\mathsf{sk}_P \leftarrow \mathsf{KeyGen}(\mathsf{MSK}, P)\}$. It sends public key, functional keys and challenge ciphertext to $\mathcal{A}$.

$\mathsf{Hyb}_{1,b}$: In this hybird, we change how the functional keys are generated for each query. The challenger chooses a key K_2 from $\mathcal{K}$ for PRF_2 and computes $(\widehat{D}_b, \mathsf{GRAM.sk}_b) \leftarrow \mathsf{GRAM.GrbDB}(1^\lambda, T, D_b)$ at the very beginning, then for each query P_i, where $i \in [Q]$

1. Sample a random string $\tau \leftarrow \{0,1\}^\lambda$.
2. Compute $\widehat{P} = \mathsf{GRAM.GrbProg}(\mathsf{GRAM.sk}_b, P; \mathsf{PRF}_1(K_1, \tau))$.
3. Set $r = \widehat{P} \oplus \mathsf{PRF}_2(K_2, \tau)$.
4. Compute and output functional key $\mathsf{sk}_P = \widetilde{\mathsf{FE}}.\mathsf{KeyGen}(\mathsf{MSK}, C[P, r, \tau])$.

The indistinguishability argument of hybrid $\mathsf{Hyb}_{0,b}$ and $\mathsf{Hyb}_{1,b}$ is based on the pseudorandom property of $\mathsf{PRF}_2(K_2, \tau)$, which is not used in any other place, and the randomness of string τ.

$\mathsf{Hyb}_{2,b}$: In this hybrid, we set the $\widetilde{\mathsf{FE}}.\mathsf{CT}$ part in challenge ciphertext as

$$\widetilde{\mathsf{FE}}.\mathsf{Enc}\left(\mathsf{pk}, (0^\lambda, 0^\lambda, K_2, 1)\right)$$

The indistinguishability between hybrid $\mathsf{Hyb}_{1,b}$ and $\mathsf{Hyb}_{2,b}$ is based on the indistinguishability security of FE scheme $\widetilde{\mathsf{FE}}$, since

$$C\left[P, r, \tau\right](\mathsf{GRAM.sk}, K_1, 0^\lambda, 0) = C\left[P, r, \tau\right](0^\lambda, 0^\lambda, K_2, 1)$$

where r, τ are generated as described in hybrid $\mathsf{Hyb}_{2,b}$.

$\mathsf{Hyb}_{3,b}$: In this hybird, we change how the hardwired value τ is generated in each functional key query. Instead of computing $\widehat{P} = \mathsf{GRAM.GrbProg}(\mathsf{sk}, P; \mathsf{PRF}_1(K_1, \tau))$, we compute $\widehat{P} = \mathsf{GRAM.GrbProg}(\mathsf{sk}, P; u)$, where $u \in \{0,1\}^{\ell_1}$ is a random string.

The indistinguishability of $\mathsf{Hyb}_{2,b}$ and $\mathsf{Hyb}_{3,b}$ follows from the security of pseudorandom function PRF_1 using key K_1, which is not used anywhere else except for computing hardwired value τ.

The indistinguishability of $\mathsf{Hyb}_{3,0}$ and $\mathsf{Hyb}_{3,1}$ follows the reusable security of garbled RAM scheme GRAM and query restraint $P^{D_0} = P^{D_1}$ for program P. □

Acknowledgement

We than Shota Yamada and anonymous ASIACRYPT 2022 reviewers for improving our work. Luowen Qian is supported by DARPA under Agreement No. HR00112020023.

References

[1] Shweta Agrawal and Monosij Maitra. FE and iO for turing machines from minimal assumptions. In *Theory of Cryptography Conference*, pages 473–512, 2018.

[2] Shweta Agrawal and Ishaan Preet Singh. Reusable garbled deterministic finite automata from learning with errors. In *ICALP*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[3] Prabhanjan Ananth, Zvika Brakerski, Gil Segev, and Vinod Vaikuntanathan. From selective to adaptive security in functional encryption. In *Annual Cryptology Conference*, pages 657–677, 2015.

[4] Prabhanjan Ananth, Yu-Chi Chen, Kai-Min Chung, Huijia Lin, and Wei-Kai Lin. Delegating ram computations with adaptive soundness and privacy. In *Theory of Cryptography Conference*, pages 3–30. Springer, 2016.

[5] Prabhanjan Ananth and Abhishek Jain. Indistinguishability obfuscation from compact functional encryption. In *Annual Cryptology Conference*, pages 308–326, 2015.

[6] Prabhanjan Ananth and Alex Lombardi. Succinct garbling schemes from functional encryption through a local simulation paradigm. In *TCC*, pages 455–472, 2018.

[7] Prabhanjan Ananth and Amit Sahai. Functional encryption for turing machines. In *Theory of Cryptography Conference*, pages 125–153, 2016.

[8] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Siddartha Telang. Succinct randomized encodings and their applications. In *STOC*, 2015.

[9] Nir Bitansky, Omer Paneth, and Alon Rosen. On the cryptographic hardness of finding a nash equilibrium. In *FOCS'15*, pages 1480–1498. IEEE, 2015.

[10] Nir Bitansky and Vinod Vaikuntanathan. Indistinguishability obfuscation from functional encryption. *Journal of the ACM (JACM)*, 65(6):39, 2018.

[11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *Theory of Cryptography*, pages 253–273. Springer, 2011.

[12] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *Advances in Cryptology-ASIACRYPT 2013*, pages 280–300. Springer, 2013.

[13] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram and applications. In *Theory of Cryptography Conference*, pages 175–204. Springer, 2016.

[14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *Public-Key Cryptography–PKC 2014*, pages 501–519. Springer, 2014.

[15] Elette Boyle, Justin Holmgren, and Mor Weiss. Permuted puzzles and cryptographic hardness. In *TCC*, 2019.

[16] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *TCC*, pages 662–693, 2017.

[17] Zvika Brakerski, Alex Lombardi, Gil Segev, and Vinod Vaikuntanathan. Anonymous IBE, leakage resilience and circular security from new assumptions. In *EUROCRYPT*, pages 535–564, 2018.

[18] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Adaptive succinct garbled ram or: How to delegate your database. In *Theory of Cryptography Conference*, pages 61–90. Springer, 2016.

[19] Ran Canetti and Justin Holmgren. Fully succinct garbled RAM. In *ITCS*, pages 169–178. ACM, 2016.

[20] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Indistinguishability obfuscation of iterated circuits and RAM programs. In *STOC*, 2015.

[21] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In *TCC*, pages 694–726, 2017.

[22] Angelo De Caro, Vincenzo Iovino, Abhishek Jain, Adam O'Neill, Omer Paneth, and Giuseppe Persiano. On the achievability of simulation-based security for functional encryption. In *Advances in Cryptology - CRYPTO 2013*, pages 519–535, 2013.

[23] Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Cryptography for parallel RAM from indistinguishability obfuscation. In Madhu Sudan, editor, *ITCS*, pages 179–190. ACM, 2016.

[24] Chongwon Cho, Nico Döttling, Sanjam Garg, Divya Gupta, Peihan Miao, and Antigoni Polychroniadou. Laconic oblivious transfer and its applications. In *Annual International Cryptology Conference*, pages 33–65, 2017.

[25] Kai-Min Chung and Luowen Qian. Adaptively secure garbling schemes for parallel computations. In Dennis Hofheinz and Alon Rosen, editors, *TCC*, 2019.

[26] Sanjam Garg, Craig Gentry, Shai Halevi, and Mariana Raykova. Two-round secure MPC from indistinguishability obfuscation. In *TCC*, pages 74–94, 2014.

[27] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *FOCS 2013*, pages 40–49, 2013.

[28] Sanjam Garg, Divya Gupta, Peihan Miao, and Omkant Pandey. Secure multiparty ram computation in constant rounds. In *TCC*, pages 491–520, 2016.

[29] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. In Venkatesan Guruswami, editor, *FOCS*, pages 210–229. IEEE, 2015.

[30] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In *STOC'15*, pages 449–458. ACM, 2015.

[31] Sanjam Garg, Rafail Ostrovsky, and Akshayaram Srinivasan. Adaptive garbled ram from laconic oblivious transfer. In *CRYPTO*, pages 515–544, 2018.

[32] Sanjam Garg, Omkant Pandey, and Akshayaram Srinivasan. Revisiting the cryptographic hardness of finding a nash equilibrium. In *Annual International Cryptology Conference*, pages 579–604, 2016.

[33] Sanjam Garg and Akshayaram Srinivasan. A simple construction of iO for Turing machines. In *TCC*, pages 425–454, 2018.

[34] Sanjam Garg and Akshayaram Srinivasan. Adaptively secure garbling with near optimal online complexity. In *EUROCRYPT*, pages 535–565, 2018.

[35] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *EUROCRYPT*, pages 405–422, 2014.

[36] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private RAM computation. In *FOCS*, pages 404–413, 2014.

[37] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 1986.

[38] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[39] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *STOC'13.*, pages 555–564, 2013.

[40] Rishab Goyal, Sam Kim, Nathan Manohar, Brent Waters, and David J Wu. Watermarking public-key cryptographic primitives. In *Annual International Cryptology Conference*, pages 367–398, 2019.

[41] Ariel Hamlin, Justin Holmgren, Mor Weiss, and Daniel Wichs. On the plausibility of fully homomorphic encryption for rams. In *CRYPTO*, 2019.

[42] Ariel Hamlin, Rafail Ostrovsky, Mor Weiss, and Daniel Wichs. Private anonymous data access. In *EUROCRYPT*, pages 244–273, 2019.

[43] Marcel Keller and Avishay Yanai. Efficient maliciously secure multiparty computation for ram. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 91–124. Springer, 2018.

[44] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 669–684. ACM, 2013.

[45] Fuyuki Kitagawa, Ryo Nishimaki, Keisuke Tanaka, and Takashi Yamakawa. Adaptively secure and succinct functional encryption: Improving security and efficiency, simultaneously. In *CRYPTO*, pages 521–551, 2019.

[46] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In *STOC*, 2015.

[47] Lucas Kowalczyk, Tal Malkin, Jonathan Ullman, and Daniel Wichs. Hardness of non-interactive differential privacy from one-way functions. In *Annual International Cryptology Conference*, pages 437–466, 2018.

[48] Huijia Lin and Stefano Tessaro. Indistinguishability obfuscation from trilinear maps and block-wise local prgs. In *CRYPTO*, pages 630–660, 2017.

[49] Qipeng Liu and Mark Zhandry. Decomposable obfuscation: a framework for building applications of obfuscation from polynomial hardness. In *Theory of Cryptography Conference*, pages 138–169. Springer, 2017.

[50] Steve Lu and Rafail Ostrovsky. Black-box parallel garbled RAM. In *CRYPTO*, 2017.

[51] Adam O'Neill. Definitional issues in functional encryption. *IACR Cryptology ePrint Archive*, 2010:556, 2010.

[52] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In *Advances in Cryptology - EUROCRYPT 2005*, pages 457–473, 2005.

[53] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In David B. Shmoys, editor, *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 475–484. ACM, 2014. URL: http://doi.acm.org/10.1145/2591796.2591825, doi:10.1145/2591796.2591825.

[54] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.