Cryptography with Updates

Prabhanjan Ananth
UCLA
prabhanjan@cs.ucla.edu

Aloni Cohen
MIT
aloni@mit.edu

Abhishek Jain
Johns Hopkins University
abhishek@cs.jhu.edu

Abstract

Starting with the work of Bellare, Goldreich and Goldwasser [CRYPTO'94], a rich line of work has studied the design of updatable cryptographic primitives. For example, in an updatable signature scheme, it is possible to efficiently transform a signature over a message into a signature over a related message without recomputing a fresh signature.

In this work, we continue this line of research, and perform a systematic study of updatable cryptography. We take a unified approach towards adding updatability features to recently studied cryptographic objects such as attribute-based encryption, functional encryption, witness encryption, indistinguishability obfuscation, and many others that support non-interactive computation over inputs. We, in fact, go further and extend our approach to classical protocols such as zero-knowledge proofs and secure multiparty computation.

To accomplish this goal, we introduce a new notion of *updatable randomized encodings* that extends the standard notion of randomized encodings to incorporate updatability features. We show that updatable randomized encodings can be used to generically transform cryptographic primitives to their updatable counterparts.

We provide various definitions and constructions of updatable randomized encodings based on varying assumptions, ranging from one-way functions to compact functional encryption.

Contents

1 Introduction

The last decade has seen the advent of a vast array of advanced cryptographic primitives such as attribute-based encryption [SW05, GPSW06], predicate encryption [BW07, SBC$^+$07, KSW08, GVW15a], fully homomorphic encryption [Gen09], fully homomorphic signatures [ABC$^+$07, BF11, GVW15b], functional encryption [SW05, BSW11, O'N10, GGG$^+$14], constrained pseudorandom functions [BW13, BGI14, KPTZ13], witness encryption [GGSW13, GLW14], witness PRFs [Zha16], indistinguishability obfuscation [BGI$^+$01, GGH$^+$13], and many more. Most of these primitives can be viewed as "cryptographic circuit compilers" where a circuit C can be compiled into an encoding $\langle C \rangle$ and an input x can be encoded as $\langle x \rangle$ such that they can be evaluated together to compute $C(x)$. For example, in a functional encryption scheme, circuit compilation corresponds to the key generation process whereas input encoding corresponds to encryption. Over the recent years, cryptographic circuit compilers have revolutionized cryptography by providing non-interactive means of computing over inputs/data.

A fundamental limitation of these circuit compilers is that they only support *static* compilation. That is, once a circuit is compiled, it can no longer be modified. In reality, however, compiled circuits may need to undergo several updates over a period of time. For example, consider an organization where each employee is issued a decryption key SK_P of an attribute-based encryption scheme where the predicate P corresponds to her access level determined by her employment status. However, if her employment status later changes, then we would want to update the predicate P associated with her decryption key. Known schemes, unfortunately, do not support this ability.

Motivated by the necessity of supporting updates in applications, in this work, we study and build *dynamic* circuit compilers. In a dynamic circuit compiler, it is possible to update a compiled circuit $\langle C \rangle$ into another compiled circuit $\langle C' \rangle$ by using an *encoded update string* whose size only depends on the "difference" between the plaintext circuits C and C'. For example, if the difference between C and C' is simply a single gate change, then this should be reflected in the size of the encoded update. Note that this rules out the trivial solution of simply releasing a new compiled circuit at the time of update.

Background: Incremental Cryptography. The study of cryptography with updates was initiated by Bellare, Goldreich and Goldwasser [BGG94] under the umbrella of *incremental cryptography*. They studied the problem of incremental digital signatures, where given a signature of a message m, it should be possible to efficiently compute a signature of a related message m', without having to recompute the signature of m' from scratch. Following their work, the study of incremental cryptography was extended to other basic cryptographic primitives such as encryption and hash functions [BGG94, BGG95, Mic97, Fis97, BM97, BKY01, MPRS12], and more recently, indistinguishability obfuscation [GP15, AJS15b].

Our Goal. In this work, we continue this line of research, and perform a systematic study of updatable cryptographic primitives. We take a unified approach towards adding updatability features to recently studied primitives such as attribute-based encryption, functional encryption and more generally, cryptographic circuit compilers. We, in fact, go further and also study updatability for classical protocols such as zero-knowledge proofs and secure multiparty computation.

To accomplish this goal, we introduce a new notion of *updatable randomized encodings* that extends the standard notion of randomized encoding [IK00] to incorporate updatability features. We show that updatable randomized encodings can be used to generically transform cryptographic primitives (discussed above) to their updatable counterparts.

Updatable Randomized Encodings. The notion of randomized encoding [IK00] allows one to encode a "complex" computation $C(x)$ into a "simple" randomized function $\mathsf{Encode}(C, x; r)$ such that given its output $\langle C(x)\rangle$, it is possible to evaluate a public Decode algorithm to recover the value $C(x)$ without learning anything else about C and x. The typical measure of "simplicity" studied in the literature dictates that the parallel-time complexity of the Encode procedure be smaller than that of computing $C(x)$. Such randomized encodings are known to exist for general circuits based on only one-way functions [AIK07] (also referred to as Yao's garbled circuits [Yao86], where the encoding complexity is in $\mathbf{NC}^1$).

In this work, we study *updatable* randomized encodings (URE): given a randomized encoding $\langle C(x)\rangle$ of $C(x)$, we want the ability to update it to an encoding $\langle C'(x')\rangle$ of $C'(x')$, where C' and x' are derived from C and x by applying some update $\mathbf{u}$. We require that the update $\mathbf{u}$ can be encoded as $\langle \mathbf{u}\rangle$ which can then be used to transform $\langle C(x)\rangle$ into $\langle C'(x')\rangle$, a randomized encoding of $C'(x')$. A bit more precisely, a URE scheme consists of the following algorithms:

- $\mathsf{Encode}(C, x)$ takes as input a circuit C and input x, and outputs an encoding $\langle C(x)\rangle$ and secret state st.
- $\mathsf{GenUpd}(\mathsf{st}, \mathbf{u})$ taking as input an update $\mathbf{u}$, and outputting an encoded update $\langle \mathbf{u}\rangle$ and a possibly updated state st'.
- $\mathsf{ApplyUpd}\left(\langle C(x)\rangle, \langle \mathbf{u}\rangle\right)$ taking as input a randomized encoding $\langle C(x)\rangle$ and an update encoding $\langle \mathbf{u}\rangle$, and outputting an updated randomized encoding $\langle C'(x')\rangle$.
- $\mathsf{Decode}\left(\langle C(x)\rangle\right)$, taking as input a (possibly updated) randomized encoding $\langle C(x)\rangle$, and outputting the value $y = C(x)$.

The key efficiency requirement is that the running time of the GenUpd algorithm must be a fixed polynomial in the security parameter and the size of the update, and independent of the size of the circuit and input being updated. This, in particular, implies that the size of an update encoding $\langle \mathbf{u}\rangle$ is also a fixed polynomial in the security parameter and the size of $\mathbf{u}$.

We define correctness and security of URE for the case of *multiple updates*. Let $\langle C_0(x_0)\rangle$ denote an initial randomized encoding. Let $\mathbf{u}_1, \ldots, \mathbf{u}_n$ denote a sequence of updates and let $\langle \mathbf{u}_i\rangle$ denote an encoding of $\mathbf{u}_i$. In a URE scheme for multiple updates, $\langle C_0(x_0)\rangle$ can be updated to $\langle C_1(x_1)\rangle$ using $\langle u_1\rangle$. The updated randomized encoding $\langle C_1(x_1)\rangle$ can then be updated into $\langle C_2(x_2)\rangle$ using $\langle u_2\rangle$, and so on, until we obtain $\langle C_n(x_n)\rangle$. We allow n to be an arbitrary polynomial in the security parameter.

To define security, we can consider two notions:

I. URE with multiple evaluations: In the first definition, each updated encoding $\langle C_i(x_i\rangle$ can be decoded to obtain $C_i(x_i)$. The security requirement is that given an initial randomized encoding $\langle C_0(x_0)\rangle$ and a sequence of updates $\{\langle \mathbf{u}_i\rangle\}_{i=1}^n$, an adversary can only learn the outputs $\{C_i(x_i)\}_{i=0}^n$, and nothing else. Here, for every $i \in [n]$, (C_i, x_i) is derived from (C_{i-1}, x_{i-1}) using the ith update $\mathbf{u}_i$.

II. URE with single evaluation: We also consider a weaker definition where *only the final encoding $\langle C_n(x_n)\rangle$ can be decoded.* To enable this, we can consider an augmented decoding algorithm that additionally requires an "unlocking key." We provide this unlocking key after all the updates are completed s.t. it allows the user to decode the final encoding and prevents her from decoding any previous encodings. More specifically, the security requirement is that given an initial randomized encoding $\langle C_0(x_0)\rangle$ and a sequence of updates $\{\langle \mathbf{u}_i\rangle\}_{i=1}^n)$, an adversary can only learn the final output $C_n(x_n)$, and nothing else.

For convenience of presentation, we in fact consider an alternative but equivalent formulation of single-evaluation URE that we refer to as **updatable garbled circuits** (UGC). Recall that a garbled circuit [Yao86] is a decomposable randomized encoding where a circuit C and an input x can be encoded separately. In an updatable garbled circuit scheme, given an encoding $\langle C_0\rangle$ of a circuit C_0 and a sequence of update encodings $\langle \mathbf{u}_1\rangle, \ldots, \langle \mathbf{u}_n\rangle$, it is possible to compute updated circuit encodings $\langle C_1 \rangle, \ldots, \langle C_n \rangle$, where C_i is derived from C_{i-1} using $\mathbf{u}_i$. Once all the updates are completed, an encoding $\langle x \rangle$ for an input x is released. This input encoding can then be used to decode the final circuit encoding $\langle C_n \rangle$ and learn $C_n(x_n)$. Intuitively, the input encoding can be viewed as the unlocking key in single-evaluation URE.

It is easy to see that UGC is a weaker notion than multi-evaluation URE. In particular, since UGC only allows for decoding "at the end," it remains *single-use*, while multi-evaluation URE captures *reusability*.

We view our notions of URE and UGC to be of independent interest from a purely complexity-theoretic perspective. Further, as we discuss later, they have powerful applications to updatable cryptography.

URE vs Reusable Garbled Circuits. Note that our definition of URE allows for *sequential* updates. One may also consider an alternative model of *parallel* updates, where each update $\langle \mathbf{u}_i \rangle$ is applied in parallel to the original encoding $\langle C_0(x_0) \rangle$. It turns out that URE with parallel updates implies the notion of reusable garbled circuits defined by Goldwasser et al [GKP+13]. In fact, we can also show a reverse implication from reusable garbled circuits to URE with parallel updates. We refer the reader to Appendix A for further discussion on this subject.

1.1 Our Results

In this work, we initiate the study of updatable randomized encodings. We study both simulation and indistinguishability-based security definitions and obtain general positive results. We showcase URE as a central object for the study of updatable cryptography by demonstrating applications to other updatable cryptographic primitives. The technical ideas we develop for our constructions are quite general, and may be applicable to future works on updatable cryptography.

I. URE for General Updates. We first state our results for multi-evaluation URE for general circuits. We allow for a general family of updates, i.e., an update can modify the circuit arbitrarily, with the only restriction that the size of the circuit remains unchanged. Below, we separately consider the case of unbounded updates and (a priori) bounded updates separately.

URE with Unbounded Updates from Secret-key Compact FE. Our first result is a construction of multi-evaluation URE for general circuits that supports an unbounded polynomial number of sequential updates. The underlying assumption is a secret-key compact functional encryption scheme for general circuits that supports a single function key query.

Theorem 1 (Informal). *Let $\mathcal{C}$ be a family of general circuits where each circuit is of the same size. Let $\mathcal{U}$ be a family of general updates with a public update algorithm* Update *s.t. for any $C \in \mathcal{C}$, input x for C, $\mathbf{u} \in \mathcal{U}$, and $(C', x') \leftarrow \mathsf{Update}(C, x, \mathbf{u})$, we have that $C' \in \mathcal{C}$. Assuming the existence of a secret-key compact functional encryption scheme for general circuits that supports a single key query, there exists a multi-evaluation* URE *scheme for $\mathcal{C}$ that supports an arbitrary polynomial number of sequential updates from $\mathcal{U}$.*

A compact functional encryption is one where the running time of the encryption algorithm for a message m is a fixed polynomial in the size of m and the security parameter, and independent of the complexity of the function family supported by the FE scheme. A recent work of Bitansky et al. [BNPW16] shows that secret-key compact functional encryption implies exponentially-efficient indistinguishability obfuscation (XIO) [LPST16a]. Put together with the results of [LPST16a] and [AJ15, BV15a], it shows that *sub-exponentially* secure secret-key compact FE that supports a single function key query together with the learning with errors (LWE) assumption implies indistinguishability obfuscation.

In contrast, in Theorem 1, we require secret-key compact FE with only *polynomial security*. Such an FE scheme can be based on polynomial-hardness assumptions on multilinear maps using the results of [GGHZ14] and [BV15a, AJS15a].

URE with Bounded Updates from One-Way Functions. For the case of polynomially bounded updates, we can, in fact, relax our assumption to only *one-way functions*. We obtain this result by using a single-key compact secret-key FE scheme for an a priori *bounded* number of ciphertexts that is constructed from one-way functions [SS10, GVW12].

To the best of our knowledge, such an FE scheme has not been explicitly stated in the literature. However, it follows easily from prior work. Very roughly, a modified version of [SS10] FE scheme where the encryption and key generation algorithms are "flipped" yields a compact secret-key FE scheme with security for a single ciphertext based on one-way functions. This can then be amplified to achieve security for polynomially bounded number of ciphertexts by applying a "flipped" version of the transformation in [GVW12], without adding any further assumptions and preserving the compactness of ciphertexts. We note, however, that the size of each ciphertext in this scheme grows with the total number of ciphertexts supported by the scheme.

Plugging in such an FE scheme in Theorem 1 yields a URE scheme for an a priori bounded polynomial number of updates.

Theorem 2 (Informal). *Let $\mathcal{C}$ and $\mathcal{U}$ be as in Theorem 1. Assuming one-way functions, for any fixed polynomial p, there exists a multi-evaluation* URE *scheme for $\mathcal{C}$ that supports p sequential updates from $\mathcal{U}$. The size of each update grows with the total number of updates p.*

An update encoding in the above construction consists of an FE encryption of the plaintext update. Since the FE scheme obtained from [SS10, GVW12] (as described above) has ciphertexts whose size depends on the total number of ciphertexts supported by the scheme, our construction inherits this inefficiency in the size of an update.

From URE to Compact FE. We next investigate whether our assumption of secret-key compact FE is necessary for building multi-evaluation URE with unbounded updates. In this regard, we show that if a (multi-evaluation) URE scheme is *output compact*, then it implies XIO. Put together with the result of [LPST16a], we have that a URE scheme with output compactness together with LWE implies a public-key compact FE scheme that supports a single key query.

Theorem 3 (Informal). *Assuming LWE, a multi-evaluation* URE *scheme with unbounded output-compact updates implies a public-key compact FE scheme that supports a single key query.*

In an output-compact URE scheme, the running time of the GenUpd algorithm is independent of the output length of the updated circuit. We remark that the URE scheme obtained from Theorem 1 is, in fact, output compact. Thus, our assumption in theorem 1 is essentially tight.

Discussion on Output Compactness. We study both indistinguishability and simulation-based security notions for URE. In the context of FE, it is known from [AGVW13, CIJ$^+$13] that simulation-secure FE with output compactness is impossible for general functions. We observe that the same ideas as in [AGVW13, CIJ$^+$13] can be used to establish impossibility of simulation-secure URE with output compact updates.

However, when we consider indistinguishability-based security, URE with output compact updates is indeed possible. The results in Theorems 1 and 2 are stated for this case. Furthermore, using the trapdoor circuits technique of [CIJ$^+$13], one can generically transform output-compact URE with indistinguishability security to non-output-compact URE with simulation-based security.

II. UGC for Bit-wise Updates. We next state our results for single-evaluation URE, a.k.a updatable garbled circuits. We consider the family of bit-wise updates, where an update

u can modify any number of bits in the bit-wise representation of a circuit, such that the size of **u** grows with the number of bits being updated. Below, we consider the case of unbounded updates and bounded updates separately.

UGC with Unbounded Updates from Lattice Assumptions. Our first result is a construction of UGC for general circuits that supports an unbounded number of sequential updates from the family of bit-wise updates. We build such a scheme from worst-case lattice assumptions.

Theorem 4 (Informal). *Let $\mathcal{C}$ be a family of general circuits and let $\mathcal{U}$ be a family of bit-wise updates. Assuming the hardness of approximating either GapSVP or SIVP to within sub-exponential factors, there exists a* UGC *scheme for $\mathcal{C}$ that supports an unbounded polynomial number of sequential updates from $\mathcal{U}$.*

At the heart of this result is a new notion of *puncturable symmetric proxy re-encryption scheme* that extends the well-studied notion of proxy re-encryption [BBS98]. In a symmetric proxy re-encryption scheme, for any pair of secret keys SK_1, SK_2, it is possible to construct a re-encryption key $\mathsf{RK}_{1\to2}$ that can be used to publicly transform a ciphertext w.r.t. SK_1 into a ciphertext w.r.t. SK_2. In our new notion of puncturable proxy re-encryption, re-encryption keys can be "disabled" on ciphertexts CT^* (w.r.t. SK_1) s.t. the semantic security of CT^* holds even if the adversary is given the punctured key $\mathsf{RK}_{1\to2}^{\mathsf{CT}^*}$ and SK_2. We give a construction of such a scheme based on the hardness of approximating either GapSVP or SIVP to within sub-exponential factors.

Given the wide applications of proxy re-encryption (see, e.g., [AFGH05] for a discussion), we believe that our notion of puncturable proxy re-encryption is of independent interest and likely to find new applications in the future. Later in this section, we discuss one such application to oblivious RAM with forward security.

UGC with Bounded Updates from One-Way Functions. For the case of polynomially bounded updates, we can relax our assumption to only *one-way functions*. We obtain this result by using a puncturable PRF scheme that can be based on one-way functions [GGM86, SW14].

Theorem 5 (Informal). *Let $\mathcal{C}$ and $\mathcal{U}$ be as in Theorem 4. Assuming one-way functions, for any fixed polynomial p, there exists a* UGC *scheme for $\mathcal{C}$ that supports p sequential updates from $\mathcal{U}$. The size of the initial garbled circuit as well as each update encoding is independent of p. However, the initial circuit garbling time and update generation time grows with p.*

The construction of this scheme is quite simple and does not require a puncturable proxy re-encryption scheme. We provide an informal description of this scheme in the technical overview section 1.2.1.

III. Applications. We next discuss applications of our results and the underlying tools.

Updatable Primitives with IND security. We start by discussing application of multi-evaluation URE to dynamic circuit compilers. Here, we demonstrate our main idea by a concrete example, namely, by showing how to use URE to transform any (key-policy) attribute-based encryption (ABE) scheme into *updatable ABE*. The same idea can be used in a generic way to build dynamic circuit compilers and obtain updatable functional encryption, updatable indistinguishability obfuscation, and so on. We refer the reader to Section 6.1 for the general case.

We briefly describe a generic transformation from any ABE scheme to one where the policies associated with secret keys can be updated. The setup and encryption algorithms for the updatable ABE scheme are the same as in the underlying ABE scheme. The key generation algorithm in the updatable ABE scheme works as follows: to compute an attribute key for a function f, we compute a URE $\langle C_f \rangle$ of a circuit C_f where C runs the key generation algorithm of the underlying ABE scheme using function f and outputs a key SK_f. To decrypt a ciphertext, a user can first decode $\langle C_f \rangle$ to compute SK_f and then use it to decrypt the ciphertext.

In order to update an attribute key for a function f to another key for function f', we can simply issue an update encoding $\langle \mathbf{u} \rangle$ for $\langle C_f \rangle$ where $\mathbf{u}$ captures the modification from f to f'. To compute the updated attribute key, a user can first update $\langle C_f \rangle$ using $\langle \mathbf{u} \rangle$ to obtain $\langle C_{f'} \rangle$, and then decode it to obtain an attribute key $\mathsf{SK}_{f'}$ for f'.

Let us inspect the efficiency of updates in the above updatable ABE scheme. As in URE, we would like the size (as well as the generation time) of an update encoding here to be independent of the size of the updated function. Note, however, that the output of the updated function $C_{f'}$ is very large – an entire attribute key $\mathsf{SK}_{f'}$! Thus, in order to achieve the aforementioned efficiency, we require that the URE scheme has updates with output compactness.

Recall that URE with output compact updates is only possible with indistinguishability-based security. As such, the above idea is only applicable to cryptographic primitives with indistinguishability-based security.

Updatable Primitives with SIM security. Next, we discuss applications of URE to cryptographic primitives with simulation-based security. In the main body of the paper, we describe two concrete applications, namely, *updatable non-interactive zero-knowledge proofs* (UNIZK) and *updatable multiparty computation* (UMPC). A notable feature of these constructions is that they only require a URE scheme with *non-output-compact* updates and simulation-based security. Below, we briefly describe our main idea for constructing UNIZKs.

Let (x, w) denote an instance and witness pair for an **NP** language L. Let $\mathbf{u}$ denote an update that transforms (x, w) to another valid instance and witness pair (x', w'). In a UNIZK proof system for L, it should be possible for a prover to efficiently compute an encoding $\langle \mathbf{u} \rangle$ of $\mathbf{u}$ that allows a verifier to transform a valid proof π for x into a proof π' for x' and verify its correctness.

We now briefly describe our transformation. A proof π for (x, w) in the UNIZK scheme is computed as follows: we first compute a URE $\langle C_{x,w} \rangle$ for a circuit $C_{x,w}$ that checks whether (x, w) satisfies the **NP** relation associated with L and outputs 1 or 0 accordingly. Furthermore, we also compute a regular NIZK proof ϕ to prove that $\langle C_{x,w} \rangle$ is computed "honestly." To verify $\pi = (\langle C_{x,w} \rangle, \phi)$, a verifier first verifies ϕ and if the check succeeds, it decodes $\langle C_{x,w} \rangle$ and outputs its answer.

In order to update a proof π, we can simply issue an update encoding $\langle \mathbf{u} \rangle$ for the randomized encoding $\langle C_{x,w} \rangle$, along with a regular NIZK proof ϕ' that $\langle \mathbf{u} \rangle$ was computed honestly. Upon receiving the update $(\langle \mathbf{u} \rangle, \phi')$, a verifier can first verify ϕ' and then update $\langle C_{x,w} \rangle$ using $\langle \mathbf{u} \rangle$ to obtain $\langle C_{x',w'} \rangle$. Finally, it can decode the updated URE $\langle C_{x',w'} \rangle$ to learn whether x' is in the language L or not.

It should be easy to see that the above idea can, in fact, be also used to make *interactive* zero-knowledge proofs updatable. Finally, we note that the above is a slightly oversimplified description and we refer the reader to Sections 6.2 and 6.3 for further details on UNIZK and UMPC, respectively.

Forward-Secure Oblivious RAM. Our notion of puncturable proxy re-encryption, when supplemented with a stronger efficiency property, can be used to build an oblivious RAM (ORAM) scheme [Gol87, Ost90, GO96] with strong *forward security* [Gün89, DvOW92, BM99]. Recall that in an ORAM scheme, the database is initially encrypted using a standard semantically secure encryption scheme with the secret-key known only to the client. Roughly speaking, in a forward secure ORAM, we want the ability to "evolve" the secret-key with every update (i.e., a read or write operation) such that if the secret key SK_i after the ith update is leaked, then all the previous updates $< i$ remain hidden from the adversary.

We now describe our idea for constructing forward secure ORAM starting from any ORAM scheme where the database is encrypted in a bit-by-bit (or block-by-block) manner. We note that this requirement holds for all known ORAM schemes. Now, suppose that we instantiate such an ORAM scheme with a puncturable re-encryption scheme as opposed to standard encryption. We now describe how the client can perform a write operation. For simplicity of exposition, let we assume that a write query consists of a single block update, but it should be easy to see that the solution extends to the general case.

Suppose that the ith write query of the client corresponds to updating the ℓth database block. Let CT_{i-1}^{ℓ} be the ciphertext corresponding to that block stored at the server, where CT_{i-1}^{ℓ} is an encryption w.r.t. secret key SK_{i-1}. Now, in order to perform the write operation, the client releases a new ciphertext CT_{i}^{ℓ} along with a punctured re-encryption key $\mathsf{RK}_{i-1,i}^{\mathsf{CT}_{i-1}^{\ell}}$. Upon receiving this, the server can replace CT_{i-1}^{ℓ} with CT_{i}^{ℓ}, and use the punctured re-encryption key $\mathsf{RK}_{i-1,i}^{\mathsf{CT}_{i-1}^{\ell}}$ to transform all the other ciphertexts from encryptions w.r.t. SK_{i-1} to encryptions w.r.t. SK_{i}.

Let us first inspect the security of this scheme. Clearly, since we are starting with an ORAM scheme, the resulting scheme is also an ORAM. More importantly, however, by using the puncturable re-encryption scheme, we achieve a strong forward security property where if the secret key SK_i after the ith update query is leaked, then all the previous updates $< i$ remain hidden from the adversary.

Finally, note that in order to achieve non-trivial ORAM efficiency, we would require that re-encrypting n ciphertext blocks takes time only sublinear (and ideally, only poly-logarithmic) in n. Our construction of puncturable re-encryption does not satisfy this property; however, it seems conceivable that such re-encryption schemes may exist. We leave this as an intriguing open question for future research.

1.2 Our Techniques

We start with the construction of UGC and present the main ideas underlying the construction. We then build upon the intuition developed in the construction of UGC, to construct (multi-evaluation) URE.

1.2.1 Construction of UGC

A *Lock-and-Release* Mechanism for Single Update. Let us first start with the simpler goal of building a UGC scheme that supports a *single* bit-wise update. Consider a universal circuit U that takes two inputs: the first input corresponds to a binary representation of a circuit C and the second input corresponds to a string x. On an input pair (C, x), U outputs $C(x)$. Our starting idea towards a UGC scheme with single update is as follows: in order to garble a circuit C, we simply compute a garbling $\tilde{U}$ (using a standard garbling scheme such as [Yao86]) of the universal circuit U and output this along with encryptions of every wire key $w_{C_1}, \ldots, w_{C_n}$ corresponding to the circuit $C = C_1, \ldots, C_n$.

Now, suppose we wish to update garbling of C to garbling of C' where C' only differs from C in the first bit. That is, $C'_i = C_i$ for every $i \neq 1$. Then, a natural idea to implement this is to release a decryption key that only decrypts the ciphertexts corresponding to the wire-keys $w_{C_2}, \ldots, w_{C_n}$ (but not w_{C_1}), along with the wire key $w_{C'_1}$ in the clear. Using this information, the receiver can recover the wire keys $w_{C'} = (w_{C'_1}, \ldots, w_{C'_n})$ (recall that $C'_i = C_i$ for every $i \neq 1$). After this update, the garbler can release the input wire keys $w_x = (w_{x_1}, \ldots, w_{x_k})$ for any input $x = x_1, \ldots, x_n$. Using all of this information, the receiver can evaluate the garbled circuit $\tilde{U}$ using the wire keys $w_{C'}$ and w_x to learn $C'(x)$.

The main remaining question is how to implement the aforementioned *conditional decryption* mechanism. A naive way to achieve this is to encrypt each wire key w_{C_i} using a fresh encryption key and then release the decryption key for every position $i \neq 1$. However, note that in this naive solution, the size of the update encoding is proportional to n, i.e., the length of the (binary representation of the) circuit C. This violates our efficiency requirement from UGC. Instead, we want that the size of an update encoding $\langle \mathbf{u} \rangle$ for an update $\mathbf{u}$ only grows with the size of $\mathbf{u}$ and the security parameter.

In order to achieve this desired efficiency, our idea is to instead use a *puncturable* encryption scheme where it is possible to compute punctured decryption keys that allow for conditional decryption of the above form. Roughly speaking, we require that for any ciphertext CT, it is possible to compute a decryption key SK_{CT} that allows one to decrypt all ciphertexts except CT. From an efficiency viewpoint, we require that the size of $SK_{\vec{\mathsf{CT}}}$ that is punctured on a set of ciphertexts $\vec{\mathsf{CT}}$ only depends on $|\vec{\mathsf{CT}}|$ (and the security parameter). Such an encryption scheme can be built from puncturable pseudorandom functions [SW14, BGI14, BW13, KPTZ13] (c.f. Waters [Wat15]) which in turn can be based on one-way functions. It is easy to verify that given such an encryption scheme, we can achieve the desired efficiency in the above construction for a single update.

We find it instructive to abstract the above idea as a **lock-and-release** mechanism. Roughly speaking, the encryption of the wire keys corresponding to C constitutes the *locking* step, while the dissemination of the punctured decryption key constitutes the (conditional) *release* step. We find this abstraction particularly useful going forward, in order to develop our full solution for an unbounded number of updates.

Multiple Updates: Main Challenges. Unfortunately, the above solution completely breaks down as soon as we consider multiple updates. In fact, the above solution does not offer any security even for two updates. This is for multiple reasons: first, the above solution allows the adversary to learn both the wire keys w_{C_i} and $w_{\bar{C}_i}$ if C_i is updated twice. The security of a standard garbling scheme completely breaks down in this case. Secondly, the above scheme does not "connect" the two updates in any manner. In particular, an adversary could choose to ignore the first update and simply use the second update (or vice-versa), or she could apply both the updates, but in the *wrong* order.

A *Layered Lock-and-Release* Mechanism for Bounded Updates. To address the above problem, we first consider the case of an a priori *bounded* number of updates. Our solution for this case already yields Theorem 5. In a nutshell, our main idea to address the above issues for the bounded update case is to use *layered* punctured encryption, or alternatively, a **layered lock-and-release** mechanism.

Suppose we wish to handle L number of updates. Then, in the solution for single update, instead of encrypting the wire key set $w_C = \{w_{C_1}, \ldots, w_{C_n}\}$ using a punctured encryption scheme, we instead use of L "onion" layers of a punctured encryption scheme. For simplicity, let us assume that each update $\mathbf{u}_i$ changes a single bit of C. Further, for notational convenience, let us assume that $\mathbf{u}_i$ corresponds to a bit position from 1 to n and the actual update simply flips the $\mathbf{u}_i$-th bit of C. Then the initial garbling of C consists of a garbling $\tilde{U}$ of the universal circuit U (as before) and L-layer encryptions of the wire keys $w_C = \{w_{C_1}, \ldots, w_{C_n}\}$, where each layer of encryption is computed w.r.t. a fresh key corresponding to a punctured encryption scheme. Now, the encoding of the first update $\mathbf{u}_1$ simply corresponds to releasing a decryption key for the outermost encryption layer that is punctured at the $\mathbf{u}_1$-th ciphertext (out of the n ciphertexts), along with a layer $(L-1)$ encryption of w_{C_ℓ}, where $\ell = \mathbf{u}_1$. More generally, an encoding of the i-th update $\mathbf{u}_i$ corresponds to releasing a punctured decryption key for i-th encryption layer along with a layer $(i-1)$ encryption of the wire key corresponding to $C_{\mathbf{u}_i}$.

The above idea of layered (punctured) encryption ensures that the receiver cannot "skip" any update, and instead must apply all the updates one-by-one to "peel-off" all the encryption layers from the wire keys. Furthermore, since the encryption layers can only be peeled off in a prescribed order, we can now also ensure that the receiver applies the updates *in order*. Finally, after all the decryption operations, the receiver only obtains a single wire key for every bit position of the (updated) circuit, and therefore, we can rely upon the security of the underlying garbled circuit.

We now briefly argue that the above construction satisfies the efficiency properties stated in Theorem 5. We first note that punctured encryption scheme in the above construction can simply correspond to a one-time pad where the randomness for computing the ith ciphertext, for every $i \in |C|$, is generated by evaluating a puncturable PRF over the index i. The PRF key (i.e., the secret key for the punctured encryption scheme) is different for every layer. With this instantiation, note that the size of the initial garbled circuit as well as every update is independent of the total number of updates L; however, the garbling time as well as update generation time depends on L.

The main problem of the above solution is that it inherently requires the number of updates to be a priori bounded. Indeed, it is not immediately clear how to extend the above solution to an unbounded number of updates.

A *Relock-and-Eventual-Release* Mechanism for Unbounded Updates. Towards that end, our main insight is to develop a **relock-and-eventual-release** mechanism as opposed to the layered lock-and-release mechanism discussed above. That is, instead of removing a lock at every step, our idea is to *change the lock* at every step. In encryption terminology, our idea is to replace the layered encryption in the above approach with a symmetric *re-encryption* scheme [BBS98]. In a symmetric re-encryption scheme, given two encryption keys SK_1 and SK_2, it is possible to issue a re-encryption key $\mathsf{RK}_{1\to 2}$ that transforms any ciphertext w.r.t. SK_1 into a ciphertext w.r.t. SK_2. In order to allow for updates, we, require the re-encryption scheme to support key puncturing. That is, we require that it is possible to compute a punctured re-encryption key $\mathsf{RK}^{\mathsf{CT}^*}_{1\to 2}$ that allows one to transform any ciphertext w.r.t. SK_1 into a ciphertext w.r.t. SK_2, *except the ciphertext* CT^* (computed under SK_1). From a security viewpoint, we require that the semantic security of CT^* should hold even if the adversary is given $\mathsf{RK}^{\mathsf{CT}^*}_{1\to 2}$ *and the terminal secret key* SK_2. We refer to such an encryption scheme as a puncturable symmetric re-encryption scheme. While the above description only refers to a "single-hop" puncturable re-encryption scheme, we in fact consider a "multi-hop" scheme.

Armed with the above insight, we modify the previous solution template as follows: the garbling of a circuit C consists of $\tilde{U}$ as before. The main difference is that the wire keys $w_C = \{w_{C_1}, \ldots, w_{C_n}\}$ corresponding to the circuit C are now encrypted w.r.t. a puncturable re-encryption scheme. Let SK_0 denote the secret key used to encrypt the wire keys. In order to issue an update encoding for an update $\mathbf{u}_i$, we release (a) a re-encryption key $\mathsf{RK}^{\mathsf{CT}}_{i-1\to i}$ that is punctured at ciphertext CT, where CT is the encryption of w_{C_ℓ} w.r.t. SK_{i-1} and ℓ is the position associated with update $\mathbf{u}_i$, along with (b) an encryption of $w_{\bar{C}_\ell}$ w.r.t. SK_i. For the final update L, we simply release the Lth secret key SK_L.

We argue the security of the construction by using the security of the puncturable re-encryption scheme and the garbling scheme (see the technical sections for details). We note, however, that this construction does not hide the location of the updates. Indeed, the correctness of the above scheme requires the evaluator to know the locations that are being updated. To address this, we provide a generic transformation from any UGC scheme (or in fact, any URE scheme) that does not achieve update hiding into one that achieves update hiding. Our transformation uses non-interactive oblivious RAM in the same manner as in [GP15]. Finally, we note that while the above only discusses single-bit updates, our construction handles multi-bit updates as well.

The only missing piece in the above solution is a construction of a puncturable symmetric re-encryption scheme. We discuss it next.

Puncturable Symmetric Re-encryption from Worst-case Lattice Assumptions. The work of [BLMR13] constructs re-encryption schemes from *key-homomorphic PRFs,* which have the property that for all x, K_1, and K_2, $\mathsf{PRF}(K_1, x) + \mathsf{PRF}(K_2, x) = \mathsf{PRF}(K_1 + K_2, x)$, where the keys and outputs of the PRF lie in appropriate groups. A secret key for the encryption scheme is simply a PRF key, and the encryption of a message m with secret key K_1 and randomness r is $\mathsf{CT} = (r, m + \mathsf{PRF}(K_1, r))$.

A re-encryption key between between secret keys K_1 and K_2 is simply their difference: $\mathsf{RK}_{1\to 2} = K_2 - K_1$. The key-homomorphism suggests a natural way to re-encrypt ciphertexts, as $(r, m + \mathsf{PRF}(K_1, r) + \mathsf{PRF}(\mathsf{RK}_{1\to 2}, r)) = (r, m + (K_2, r))$ is a ciphertext w.r.t K_2. Observe that successful re-encryption of a ciphertext with randomness r relies on the ability to compute $\mathsf{PRF}(\mathsf{RK}_{1\to 2}, r)$.

We construct *puncturable* proxy re-encryption scheme following the above approach, but instantiated with *constrained* key-homomorphic PRFs [BV15b]. A punctured re-encryption key $\mathsf{RK}_{1\to 2}^{\mathsf{CT}^*}$ for a ciphertext CT^* with randomness r^* is the PRF key $K_2 - K_1$ punctured at the input r^*. This key, which can be used to evaluate $\mathsf{PRF}(K_2 - K_1, r)$ for all $r \neq r^*$, enables the re-encryption of all ciphertexts except for the ciphertext CT^*.

For security, we require that the semantic security of CT^* holds given both $\mathsf{RK}_{1\to 2}^{\mathsf{CT}^*}$ and K_2. We reduce to the security of the constrained PRF, which guarantees that $y^* := \mathsf{PRF}(K_2 - K_1, r^*)$ is pseudorandom. The key idea is that (partial information about) y^* can be computed given CT^*, K_2, and (partial information about) the message m.

1.2.2 Construction of URE

We now shift our focus on building multi-evaluation URE.

Relock-and-Release Mechanism. Recall that the main difference between UGC and URE is that UGC only allows for a single evaluation after a sequence of updates, while URE allows for evaluation after *every* update. As such, the relock-and-*eventual*-release mechanism that we discussed above does not suffice for building URE. Our starting insight is to instead develop a **relock-and-release** mechanism that performs both relocking and release at every step. Intuitively, relocking allows us to "carry over" the updates, while the release mechanism allows us to evaluate the updated randomized encoding at every step.

Starting Idea: Garbled RAM with Persistent Memory. A natural starting approach to implement such a relock-and-release mechanism is via the use of garbled RAMs with persistent memory [LO13, GHL$^+$14]. In a garbled RAM scheme, it is possible to encode a database D_0 and later issue encodings for RAM programs $M_1, \ldots, M_n$. Each RAM program encoding $\widetilde{M_i}$ updates the database encoding from $\widetilde{D_{i-1}}$ to $\widetilde{D_i}$, and outputs the result of some computation on D_i.

Given this description, it is not difficult to see why such a notion is useful for our purpose. Starting from a garbled RAM scheme and a standard randomized encodings scheme without updates [Yao86], we can build a candidate construction of multi-evaluation URE as follows:

- We set the initial database D_0 in garbled RAM to the initial circuit and input pair (C_0, x_0) in the URE scheme. The initial updatable randomized encoding of (C_0, x_0) is an encoding of D_0, computed under garbled RAM scheme, along with an encoding of (C_0, x_0) computed under the standard randomized encoding scheme.
- In order to compute an encoding $\langle \mathbf{u}_i \rangle$ for an update $\mathbf{u}_i$, we compute an encoding $\widetilde{M_i}$ of a machine M_i w.r.t. the garbled RAM scheme where the machine M_i has $\mathbf{u}_i$ hardcoded in it. The machine M_i on input $D_{i-1} = (C_{i-1}, x_{i-1})$ first updates the database to $D_i = (C_i, x_i)$, where $(C_i, x_i) \leftarrow \mathsf{Update}(C_{i-1}, x_{i-1}; \mathbf{u}_i)$, and outputs a fresh standard randomized encoding of (C_i, x_i).

Let us inspect the above solution closely; specifically, the complexity of the machine M_i corresponding to an update $\mathbf{u}_i$. Since M_i computes a fresh (standard) randomized encoding "on-the-fly," in order to achieve the necessary efficiency guarantee for URE, we will require that the encoding time for M_i is independent of its running time. Such a garbled RAM scheme is called a *succinct* garbled RAM scheme [BGL+15, CHJV15]. Furthermore, since the output of M_i consists of a fresh randomized encoding, we will also require that the time of encode M_i is independent of its output length. Such a garbled RAM scheme is referred to as *output-compressing* [AJ15, LPST16b].

Recent works [AJ15, LPST16b] show that output-compressing succinct garbled RAM (with sub-exponential security) imply indistinguishability obfuscation (iO). Furthermore, the only known constructions for such a garbled RAM scheme are based on iO, which, in turn seems to require sub-exponential hardness assumptions. Our goal, however, is to obtain a solution for URE using *polynomial hardness assumptions.* As such, the above is not a viable solution for us.

Garbled RAM meets Delegation of Computation. Towards that end, our next idea is to instantiate the above approach using a non-succinct garbled RAM scheme where the size of the encoding of a machine M_i depends on the running time and the output length of M_i. Such garbled RAM schemes are known to exist based on only one-way functions. At first, it is not clear how to make this approach work since the efficiency requirements of URE are immediately violated.

Towards that end, our next idea is to delegate the computation of the encoding of M_i to the receiver. We implement this idea by using secret-key functional encryption [SW05, BSW11, O'N10]. Roughly speaking, the initial encoding of $C_0(x_0)$ now corresponds to a database encoding of $D_0 = (C_0, x_0)$ w.r.t. a non-succinct garbled RAM scheme along with FE functional key for a circuit P that takes as input an update string $\mathbf{u}_i$ and outputs an encoding $\widetilde{M_i}$ of the machine M_i (as described before). Encoding of an update $\mathbf{u}_i$ now corresponds to an FE encryption of $\mathbf{u}_i$.

In order to achieve the necessary efficiency guarantee of URE, we require that the secret-key FE scheme used above is *compact*, i.e., where the running time of the encryption algorithm on a message m is a fixed polynomial in the length of m and the security parameter, and in particular, independent of the size complexity of any function f in the function family supported by the FE scheme. Indeed, if this were not the case, then the encoding time for an update $\mathbf{u}_i$ in the above solution would depend on the size of the circuit C, which in turn depends on the running time and output length of M_i. Therefore, if the FE scheme were not compact, then the efficiency requirements of URE would once again be violated.

As discussed earlier, a secret-key compact FE scheme with polynomial hardness can be built from polynomial hardness assumptions on multilinear maps using using the results of [GGHZ14] and [BV15a, AJS15a].

Challenges in Proving Security. While the above construction seems to achieve correctness, it is not immediately clear how to argue security. Note that the circuit P computed by an FE key in the above construction contains the garbling key of the garbled RAM scheme hardwired inside it. Indeed, this is necessary for it to compute the encodings corresponding to machines M_i as discussed above. In order to leverage security of garbled RAM, one approach is to remove the garbling key from the FE function key. However, in order to maintain functionality, this would require hardwiring the output of P, either in the FE key, or in the FE ciphertext. We cannot afford to hardwire the output in the ciphertext since that would violate the efficiency requirements of URE. Thus, our only option is to hardwire the output in the FE key. Note, however, that in the setting of multiple updates, we have to deal with multiple outputs. In particular, the above approach would require hardwiring all the outputs (one corresponding to each update) in the FE key. Doing so "at once" would require putting a bound on the number of updates.

A better option is to hardwire the outputs "one-at-a-time," analogous to many proofs in the iO literature (see, e.g., [GLSW14, AJ15, BV15b]). Implementing this idea, however, would require puncturing the garbling key. Such a notion of key puncturing is not supported by standard garbled RAM schemes.

Using Cascaded Garbled Circuits. Towards that end, we take a step back and revisit our requirements from the garbled RAM scheme. Our first observation is that in the above solution template, machine M_i need not be a RAM since we are already requiring it to read the entire database! Instead, the key property of garbled RAM with persistent memory that is used in the above template is its ability to maintain *updated state* in the form of encoded database.

We now discuss how to implement this property in a more direct manner by "downgrading" the garbled RAM to a cascaded garbled circuit. Along the way, we will also address the security issues discussed above. Very briefly, we modify the above construction as follows: consider a circuit Q_i that has an update string $\mathbf{u}_i$ hardwired in its description. It takes as input (C_{i-1}, x_{i-1}) and outputs two values. The first value is a fresh randomized encoding of $C_i(x_i)$ where $(C_i, x_i) \leftarrow \mathsf{Update}(C_{i-1}, x_{i-1}; \mathbf{u}_i)$, and the second value is a set of wire keys for the string (C_i, x_i) corresponding to a garbling of the circuit Q_{i+1} (that is defined analogously to Q_i). The initial encoding of $C_0(x_0)$ now corresponds to the input wire keys for the string (C_0, x_0) corresponding to a garbling of circuit Q_1 as defined above, as well as an FE key for a function f that takes as input $\mathbf{u}_i$ and outputs a garbling a circuit Q_i. The encoding of an update $\mathbf{u}_i$ now corresponds to an FE encryption of $\mathbf{u}_i$ as before.

We prove the security of the above construction with respect to indistinguishability-based security definition. Simulation-based security can be argued via a generic transformation following [CIJ+13]. Let C_0^0, C_0^1, x be the initial circuits and input submitted by the adversary in the security proof. And let, $(\mathbf{u}_1^0, \mathbf{u}_1^1), \ldots, (\mathbf{u}_q^0, \mathbf{u}_q^1)$ be the tuple of updates. There are two "chains" of updating processes with the 0^{th} chain starting from C_0^0 and 1^{st} chain starting from C_1^1. The i^{th} "bead" on 0^{th} (resp., 1^{st}) chain corresponds to update $\mathbf{u}_i^0$ (resp., $\mathbf{u}_i^1$).

In the security proof, we start with the real experiment where challenge bit 0 is used. That is, the 0^{th} chain is active in the experiment. In the next step, we introduce the 1^{st} chain, along with the already present 0^{th} chain, into the experiment. However even in this step, 0^{th} chain is still active – that is, generating the randomized encoding at every step is performed using the 0^{th} chain. In the next intermediate hybrids, we slowly switch from 0^{th} chain being activated to 1^{st} chain being activated. In the i^{th} intermediate step, the first i beads on 1^{st} chain are active and on the 0^{th} chain, all except the first i beads are active – this means that the first i updated randomized encodings are computed using the 1^{st} chain and the rest of them are computed using 0^{th} chain. At the end of these intermediate hybrids, we have the 1^{st} chain to be active and 0^{th} chain to be completely inactive. At this stage, we can remove the 0^{th} chain and this completes the proof.

The two chains described above are implemented in a sequence of garbled circuits, that we call *cascaded* garbled circuits. That is, every i^{th} garbled circuit in this sequence produces wire keys for the next garbled circuit. Every garbled circuit in this sequence is a result of $\mathsf{ApplyUpd}$ procedure and encapsulates, for some i, the i^{th} beads on both the chains. In order to move from the i^{th} intermediate step to $(i+1)^{th}$ intermediate step, we use the security of garbled circuits. But since these garbled circuits are not given directly, but instead produced by a FE key, we need to make use of security of FE to make this switch work.

1.3 Related Work

Incremental Cryptography. The area of incremental cryptography was pioneered by Bellare, Goldreich and Goldwasser [BGG94]. While their work dealt with signature schemes, the concept of incremental updates has been subsequently studied for other basic cryptographic primitives such as hash functions, semantically-secure encryption and deterministic encryption [BGG95, Mic97, Fis97, BKY01, MPRS12]. To the best of our knowledge, all of these works only consider bit-wise updates.

While our work shares much in spirit with these works, we highlight one important difference. In incremental cryptography, update operation is performed "in house," e.g., in the case of signatures, the entity who produces the original signature also performs the update. In contrast, we consider a *client-server* scenario where the client simply produces an update encoding, and the actual updating process is performed by the server. This difference stipulates different efficiency and security requirements. On the one hand, incremental cryptography necessarily requires efficient updating time for the notion to be non-trivial, while we consider the weaker property of efficient update encoding generation time. On the other hand, our security definition is necessarily stronger since we allow the adversary to view the update encodings – a property not necessary when the updating is done "in house."

Incremental/Patchable Obfuscation. Recently, [GP15] and [AJS15b] study the notion of updatability in the context of *indistinguishability obfuscation*. The work of [GP15] considers incremental (i.e., bit-wise) updates, while [AJS15b] allow for arbitrary updates, including those that may increase the size of the program (modeled as a Turing machine).

We note that one of our results, namely, URE with unbounded updates can be derived from [AJS15b] at the cost of requiring sub-exponentially secure iO. In contrast, we obtain our result by using *polynomially secure* secret-key compact FE.

Malleable NIZKs. Our notion of updatable NIZKs should be contrasted with the notion of malleable NIZKs proposed by Chase et al. [CKLM12]. In a malleable NIZK, it is possible to publicly "maul" a proof string π for a statement x into a a proof string π' for a related statement x'. In contrast, our notion of UNIZK only allows for privately generated updates. To the best of our knowledge, malleable NIZKs are only known either for a limited class of update relations from standard assumptions [CKLM12], or for general class of update relations based on non-falsifiable assumptions such as succinct non-interactive arguments [CKLM13]. In contrast, we show how to build UNIZK for unbounded number of general updates from compact secret-key FE and regular NIZKs, and for a bounded number of general updates from regular NIZKs.

Updatable Codes. The concept of updating was also studied in the context of error correcting codes by [CKO14]. In this context, it is difficult to model the problem of updating – we should be able to change few bits of the code to correspond to a codeword of a different message and at the same time we want the distance between codewords of different messages to be far apart. We refer the reader to their work for discussion on this seemingly contradictory requirement. In a subsequent work, [DSLSZ15] studied this problem in the context of non-malleable codes.

2 Preliminaries

We denote the security parameter by λ.

2.1 Randomized Encodings

The notion of randomized encodings studied by Ishai-Kushilevitz [IK00] allows for encoding a "complex" computation using a "simpler" function. The measure of simplicity varies according to the context. In our work, we consider encoding circuits where the encoding function has a smaller depth compared to the original circuit. The algorithms associated with a randomized encoding RE are:

- $\widetilde{C(x)} \leftarrow \mathsf{RE.Encode}(1^\lambda, C, x)$: On input circuit C, x, it outputs the randomized encoding $\widetilde{C(x)}$.
- $C(x) \leftarrow \mathsf{RE.Decode}\left(\widetilde{C(x)}\right)$: This is a deterministic algorithm. On input the randomized encoding $\widetilde{C(x)}$, it outputs the value $C(x)$.

The correctness property says that the output of the decode algorithm on input a valid encoding of (C, x) is $C(x)$. The efficiency property states that $\mathsf{RE.Encode}$ can be represented by a circuit that has depth "smaller" than the depth of C. Typically, we have C to be an arbitrary circuit and we require Encode to be in NC^1. We present the security definition below.

Definition 1 (Randomized Encodings)**.** *We say that* RE *is secure if there exists a simulator* Sim *such that the following holds for every circuit C, x:*

$$\{\ \mathsf{RE.Encode}(1^\lambda, C, x)\ \} \approx_c \{\ \mathsf{Sim}(1^\lambda, \phi(C), C(x))\ \},$$

where $\phi(C)$ denotes the topology of the circuit C.

Garbled Circuits. Another object of interest for our work is the notion of garbled circuits. Garbled circuits is a type of randomized encoding where the input and the circuit are encoded separately. We present a modified formalization of garbled circuits from [BHR12]. A garbled circuit scheme for a class of circuits $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda \in \mathbb{N}}$ is a tuple of algorithms $\mathsf{GC} = (\mathsf{Gen}, \mathsf{GrbCkt}, \mathsf{GrbInp}, \mathsf{EvalGC})$ has the following syntax.

- **Circuit Garbling,** $\left(\langle C \rangle_{\mathsf{gc}}, \mathsf{st}\right) \leftarrow \mathsf{GrbCkt}(1^\lambda, C \in \mathcal{C}_\lambda)$: On input security parameter λ, circuit C, output a garbled circuit $\langle C \rangle_{\mathsf{gc}}$ and state st.
- **Input Garbling,** $\langle x \rangle_{\mathsf{gc}} \leftarrow \mathsf{GrbInp}(\mathsf{st}, x \in \{0,1\}^\lambda)$: On input state st, input x, it produces an input encoding $\langle x \rangle_{\mathsf{gc}}$.
- **Evaluation,** $\alpha \leftarrow \mathsf{EvalGC}\left(\langle C \rangle_{\mathsf{gc}}, \langle x \rangle_{\mathsf{gc}}\right)$: On input garbled circuit $\langle C \rangle_{\mathsf{gc}}$, input encoding $\langle x \rangle_{\mathsf{gc}}$, output the decoded value α.

The correctness property states that the output of EvalGC, on input $(\langle C \rangle_{\mathsf{gc}}, \langle x \rangle_{\mathsf{gc}})$, is $C(x)$ if $\langle C \rangle_{\mathsf{gc}}$ is a garbled circuit C and $\langle x \rangle_{\mathsf{gc}}$ is an encoding of x computed according to the above scheme.

The efficiency requirement states that the algorithms GrbCkt and EvalGC are polynomial time in the input length and the security parameter λ, and the time to generate (and the size of) the garbled input $\langle x \rangle_{\mathsf{gc}}$ is $\text{poly}(\lambda, |x|)$ for some fixed poly. In particular, it is independent of $|C|$.

Projective Schemes: In many schemes, the st encodes $|x|$-many pairs of tokens which we call *input wire labels*, and the garbled input $\langle x \rangle_{\mathsf{gc}}$ consists of one input wire label from each pair, selected according to the value of $x_i \in \{0,1\}$. We call such schemes *projective*. More formally, there exists an algorithm $\mathsf{GrbInputWire} : (\mathsf{st}, i \in [\ell_i n]) \mapsto (L_0, L_1)$ that maps a garbling state st and an input index i to a pair of input wire labels L_0, L_1, such that

$$\mathsf{GrbInp}(\mathsf{st}, x_1, \ldots, x_{\ell_{in}}) = \mathsf{GrbInputWire}(\mathsf{st}, 1)_{x_1}, \ldots, \mathsf{GrbInputWire}(\mathsf{st}, \ell_{in})_{x_{\ell_{in}}}$$

Security. There are two security notions one can consider.

Simulation-Based Security. This security definition states that a garbled circuit of C and input encoding of x can be simulated by just knowing the output $C(x)$.

Definition 2. *A garbling scheme* $\mathsf{GC} = (\mathsf{Gen}, \mathsf{GrbCkt}, \mathsf{GrbInp}, \mathsf{EvalGC})$ *for a class of circuits* $\mathcal{C} = \{\mathcal{C}_n\}_{n\in\mathbb{N}}$ *is said to be secure if there exists a simulator* SimGrb *such that for any* $C \in \mathcal{C}_n$*, any input* $x \in \{0,1\}^n$*, the following two distributions are computationally distinguishable.*

1. $\left\{ \mathsf{SimGrb}\left(1^\lambda, \phi(C), C(x)\right) \right\}$*, where* $\phi(C)$ *denotes the topology of the circuit* C*.*
2. $\left\{ (\langle C\rangle_{\mathsf{gc}}, \langle x\rangle_{\mathsf{gc}}) \right\}$*, where* $(\langle C\rangle_{\mathsf{gc}}, \mathsf{st}) \leftarrow \mathsf{GrbCkt}(1^\lambda, C \in \mathcal{C}_\lambda)$ *and* $\langle x\rangle_{\mathsf{gc}} \leftarrow \mathsf{GrbInp}(\mathsf{st}, x \in \{0,1\}^\lambda)$*.*

Indistinguishability-Based Security. We define the indistinguishability-based privacy guarantee $\mathsf{PrivIND}_\phi$ by means of a security game. We denote the challenger by Ch and the PPT adversary by $\mathcal{A}$.[1] The security is defined with respect to deterministic side-information function ϕ, which captures what information is leaked by the garbled circuits. For example, the side-information function in the original garbling scheme of Yao is ϕ_{topo}, the function that outputs the circuit topology, including size, number of inputs, and number of outputs.[2]

$\underline{\mathsf{Expt}^{\mathsf{GC}}_{\mathcal{A},\mathsf{Ind}}(1^\lambda, b \in \{0,1\})}$:

- $\mathcal{A}$ sends circuits $C^0, C^1 \in \mathcal{C}_\lambda$, inputs $x_0, x_1 \in \{0,1\}^\lambda$ to Ch. If $\phi_{\mathsf{topo}}(C^0) \neq \phi_C(C^1)$, abort and return $\perp$. Otherwise, Ch sends $\langle C^b\rangle_{\mathsf{gc}}$, where $(\langle C^b\rangle_{\mathsf{gc}}, \mathsf{st}) \leftarrow \mathsf{GrbCkt}(1^\lambda, C^b)$, to $\mathcal{A}$.
- If $C^0_s(x^0) \neq C^1_s(x^1)$, abort and return $\perp$. Otherwise, Ch sends $\mathsf{GrbInp}(\mathsf{st}_s, \langle x^b\rangle_{\mathsf{gc}})$ to the adversary.
- $\mathcal{A}$'s output is returned by $\mathsf{Expt}^{\mathrm{UGC}}_{\mathcal{A},\mathsf{Ind}}(1^\lambda, b)$.

Definition 3 ($\mathsf{PrivIND}_\phi$)**.** *A garbling scheme* GC *is* $\mathsf{PrivIND}_\phi$*-secure if for any PPT adversary* $\mathcal{A}$ *and some negligible function* negl

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

Yao [Yao86, LP09] presented a construction of garbled circuits based on one-way functions. Under suitable assumptions, such as DDH, LWE, the garbling algorithms in this construction can be implemented by NC^1.

2.2 Private-Key Functional Encryption

A private-key functional encryption (FE) scheme FE over a message space $\mathcal{M} = \{\mathcal{M}_\lambda\}_{\lambda\in\mathbb{N}}$ and a function space $\mathcal{F} = \{\mathcal{F}_\lambda\}_{\lambda\in\mathbb{N}}$ is a tuple $(\mathsf{FE.Setup}, \mathsf{FE.KeyGen}, \mathsf{FE.Enc}, \mathsf{FE.Dec})$ of PPT algorithms with the following properties:

- $\mathsf{FE.Setup}(1^\lambda)$: The setup algorithm takes as input the unary representation of the security parameter, and outputs a secret key $\mathsf{FE.MSK}$.
- $\mathsf{FE.KeyGen}(\mathsf{FE.MSK}, f)$: The key-generation algorithm takes as input the secret key $\mathsf{FE.MSK}$ and a function $f \in \mathcal{F}_\lambda$, and outputs a functional key $\mathsf{FE.SK}_f$.
- $\mathsf{FE.Enc}(\mathsf{FE.MSK}, m)$: The encryption algorithm takes as input the secret key $\mathsf{FE.MSK}$ and a message $m \in \mathcal{M}_\lambda$, and outputs a ciphertext CT.
- $\mathsf{FE.Dec}(\mathsf{FE.SK}_f, \mathsf{CT})$: The decryption algorithm takes as input a functional key $\mathsf{FE.SK}_f$ and a ciphertext CT, and outputs $m \in \mathcal{M}_\lambda \cup \{\perp\}$.

[1] In fact, a stronger simulation-based notion is achived, but we require only indistinguishability.
[2] For further discussion on side-information, see [BHR12].

In terms of correctness, we require that there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all sufficiently large $\lambda \in \mathbb{N}$, for every message $m \in \mathcal{M}_\lambda$, and for every function $f \in \mathcal{F}_\lambda$ it holds that

$$\mathsf{FE.Dec}(\mathsf{FE.KeyGen}(\mathsf{FE.MSK}, f), \mathsf{FE.Enc}(\mathsf{FE.MSK}, m)) = f(m)$$

with probability at least $1 - \mathsf{negl}(\lambda)$, where $\mathsf{FE.MSK} \leftarrow \mathsf{FE.Setup}(1^\lambda)$, and the probability is taken over the random choices of all algorithms.

Function privacy. We present the function privacy security definition of a secret-key FE scheme. The notion of function privacy is modeled as a game between the challenger and the adversary. In this game, a function query made by the adversary is a pair of functions and in response it receives a functional key corresponding to either of the two functions. As long as both the functions are such that they do not split the challenge message-pairs, the adversary should not be able to tell which function was used to generate the functional key. That is, the output of the left function on the left message should be the same as the output of the right function on the right message.

We incorporate the notion of function privacy in the selectively-secure FE and adaptively-secure FE definitions given earlier. First, we define the notion of function-private selective security.

Definition 4 (Function-private selectively-secure FE)**.** *A private-key functional encryption scheme* $\mathsf{FE} = (\mathsf{FE.Setup}, \mathsf{FE.KeyGen}, \mathsf{FE.Enc}, \mathsf{FE.Dec})$ *over a function space* $\mathcal{F} = \{\mathcal{F}_\lambda\}_{\lambda\in\mathbb{N}}$ *and a message space* $\mathcal{M} = \{\mathcal{M}_\lambda\}_{\lambda\in\mathbb{N}}$ *is a* ***function-private selectively-secure private-key FE scheme*** *if for any PPT adversary* $\mathcal{A}$ *there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that for all sufficiently large* $\lambda \in \mathbb{N}$*, the advantage of* $\mathcal{A}$ *is defined to be*

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

where for each $b \in \{0,1\}$ *and* $\lambda \in \mathbb{N}$ *the experiment* $\mathsf{Expt}^{\mathsf{FE}}_{\mathcal{A}}(\lambda, b)$*, modeled as a game between the challenger and the adversary* $\mathcal{A}$*, is defined as follows:*

1. *The challenger first executes* $\mathsf{FE.Setup}(1^\lambda)$ *to obtain* $\mathsf{FE.MSK}$.
2. ***Message queries:*** *The adversary submits the message-pairs* $((m_1^{(0)}, \ldots, m_q^{(0)}), (m_1^{(1)}, \ldots, m_q^{(1)}))$ *to the challenger, where* q *is the number of the message queries (which is a polynomial in the security parameter* λ*) made by the adversary. The challenger then sends* $(c_1^*, \ldots, c_q^*)$ *to* $\mathcal{A}$*, where* c_i^* *is the output of* $\mathsf{FE.Enc}(\mathsf{FE.MSK}, m_i^{(b)})$.
3. ***Function queries:*** *The adversary then makes functional key queries. For every function-pair query* (f_0, f_1)*, the challenger sends* $\mathsf{FE}.sk_{f_b}$ *to the adversary, where* $\mathsf{Ad}.sk_{f_b}$ *is the output of* $\mathsf{FE.KeyGen}(\mathsf{FE.MSK}, f_b)$*, only if* $f_0(m_i^{(0)}) = f_1(m_i^{(1)})$*, for all* $i \in [q]$*. Otherwise, it aborts.*
4. *The output of the experiment is* b'*, where* b' *is the output of* $\mathcal{A}$*.*

Compact FE. A secret keys functional encryption scheme is said to be compact if the encryption complexity is independent of the complexity of the function family. More formally,

Definition 5 ([AJ15, BV15a])**.** *Let* p *be a polynomial. A secret key FE scheme* $\mathsf{FE} = (\mathsf{FE.Setup}, \mathsf{FE.KeyGen}, \mathsf{FE.Enc}, \mathsf{FE.Dec})$ *is said to be* **compact** *if the running time of* $\mathsf{FE.Enc}(\mathsf{FE.MSK}, m)$ *is* $p(\lambda, |m|)$*, where* $\mathsf{FE.MSK} \leftarrow \mathsf{FE.Setup}(1^\lambda)$*.*

In particular, a compact FE scheme allows for generating functional keys for circuits whose size and output-length are not fixed in the setup phase.

In this work, we require a single-key secret key compact FE scheme in order to construct updatable randomized encodings. Currently, we know how to build this either from concrete assumptions on multilinear maps [GGHZ14] or based on iO [GGH+13, Wat15]. Although sub-exponentially secure, compact public-key FE is known to imply iO [AJ15, BV15a], the current approaches don't extend to the secret key setting (see [BV15a]). We emphasize that we only require *polynomially-secure, secret key* compact FE for this work.

2.3 Updatable Circuits

A boolean circuit C is an directed acyclic graph of in-degree at most 2 with the non-leaf nodes representing $\vee$ (OR), $\wedge$ (AND) and $\neg$ (NOT) gates and the leaf nodes representing the input variables and constants 0,1. The nodes with no outgoing edges are designated to be output gates. The size of a circuit $|C|$ is the number of nodes in the graph. Each node is labeled with a different index between 1 and $|C|$. The evaluation of C on input x is performed by first substituting the leaf nodes with the value x and then evaluating gate-by-gate till we reach the output gates. The joint value of all the output gates determine the output of the circuit. Circuit C is said to represent a function $f : \{0,1\}^\lambda \rightarrow \{0,1\}^{\ell(\lambda)}$ if for every $x \in \{0,1\}^\lambda$ we have $C(x) = f(x)$. We assume that the class of all boolean circuits for every fixed size $|C|$ and n inputs has an efficient binary representation $\mathsf{binary}(C) \in \{0,1\}^{O(|C|)}$. That is, there is an efficient algorithm that computes $C \mapsto (n, |C|, \mathsf{binary}(C))$, and its inverse.

We define the notion of updatable circuits next. A family of updatable circuits $\mathcal{C}$ has associated with it a class of updates $\mathcal{U}$. Given any circuit $C \in \mathcal{C}$ we can transform this circuit into another circuit $C' \in \mathcal{C}$ with the help of an update $\mathbf{u} \in \mathcal{U}$. The updating process could, for instance, change one of the output gates from $\vee$ to $\neg$, change all the gates to $\wedge$ gates and so on. Formally,

Definition 6 (Updatable Circuits)**.** *Consider a circuit family $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda \in \mathbb{N}}$, where $\mathcal{C}_\lambda$ contains* $\mathrm{poly}(\lambda)$*-sized boolean circuits $C : \{0,1\}^\lambda \rightarrow \{0,1\}^{\ell(\lambda)}$. Consider a set system of strings $\mathcal{U} = \{\mathcal{U}_\lambda\}_{\lambda \in \mathbb{N}}$, where $\mathcal{U}_\lambda$ is a set of strings of length* $\mathrm{poly}(\lambda)$*. We define $\mathcal{C}$ to be* $(\mathsf{Upd}, \mathcal{U})$**-updatable** *if $C' \leftarrow \mathsf{Upd}(C, \mathbf{u} \in \mathcal{U}_\lambda)$ is also a boolean circuit with input domain $\{0,1\}^\lambda$ and output domain $\{0,1\}^{\ell(\lambda)}$.*

The size of update $\mathbf{u}$ could potentially be much smaller than the size of the circuit C. For instance, the length of the instruction to change all the gates in C to $\wedge$ gate is in fact independent of $|C|$.

In this work, we only consider updates which change the underlying binary representation of the circuit, which we term "bitwise" updating (ex: setting the 6th bit of the circuit representation to 1). As we will see later, this in itself suffices for many interesting applications. For example, in many natural circuit representations this captures the ability to change the functionality of individual gates of the circuit. We can also consider more general updates (ex: adding gates, or changing every $\wedge$-gate occurring in even positions to $\vee$) which we leave as a direction to future exploration.

Definition 7 (Bit-wise updatable circuits)**.** *A family of circuits $\mathcal{C}$ is* **bit-wise updatable** *if it is $(\mathsf{Upd}_{\mathsf{bit}}, \mathcal{U}_{\mathsf{bit}})$-updatable, where:*

- $\mathcal{U}_{\mathsf{bit}} = \{(i,b) \in \mathbb{Z}^{\geq 0} \times \{0,1\}\}$.
- $\mathsf{Upd}_{\mathsf{bit}}(C,(i,b))$ *takes as input a circuit $C \in \mathcal{C}$ and a update $(i,b) \in \mathcal{U}_{\mathsf{bit}}$ indicating the bit index in the circuit representation $\mathsf{binary}(C)$ to be changed, and a new bit. If $i > |C|$, then $\mathsf{Upd}_{\mathsf{bit}}$ outputs $\perp$. Otherwise, output the circuit that is essentially $\mathsf{binary}(C)$ except that the i^{th} location now contains the bit b. Formally, output the circuit $\mathsf{binary}(C) \oplus 0^{i-1}\ b\ 0^{|\mathsf{binary}(C)|-i}$ which results from changing bit i of $\mathsf{binary}(C)$ to b.*

3 Updatable Randomized Encodings

We define the notion of updatable randomized encodings (URE) next. Since this notion deals with transforming circuits, this notion will be associated to a class of updatable circuits. But to also capture the joint updatability of both the circuit and the input together, we introduce the notion of hardwired circuits below.

Hardwired Circuits. A hardwired circuit, associated to a circuit C and input x, takes no input but upon evaluation yields a fixed output $C(x)$.

We provide the formal definition of hardwired circuits below.

Definition 8 (Hardwired Circuit)**.** *Consider a circuit* $C : \{0,1\}^\lambda \rightarrow \{0,1\}^{\ell(\lambda)}$ *and* $x \in \{0,1\}^\lambda$. *We define a* ***hardwired circuit****, denoted by* $C[x]$*, to be a circuit such that,*

- *it takes no input.*
- *upon evaluation (always) outputs* $C(x)$.

We interchangeably use $C[x]$ *to denote the circuit as well as the output* $C(x)$ *it computes.*

Two hardwired circuits $C_0[x_0]$ and $C_1[x_1]$ are *equivalent* if and only if $C_0(x_0) = C_1(x_1)$ and $|C_0| = |C_1|$. If $C_0[x_0]$ and $C_1[x_1]$ are equivalent then they are denoted by $C_0[x_0] \equiv C_1[x_1]$. We can generalize this notion and define a class of hardwired circuits as stated below.

Definition 9. *Consider a circuit family* $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda \in \mathbb{N}}$. *We define a* **hardwired circuit family** $\{\mathcal{C}[X]_\lambda\}_{\lambda \in \mathbb{N}}$ *where* $\mathcal{C}[X]_\lambda$ *comprises of hardwired circuits of fixed input length and is associated with a bijective function* $\phi : \mathcal{C}_\lambda \times \{0,1\}^\lambda \rightarrow \mathcal{C}[X]_\lambda$ *such that if* $\phi(C \in \mathcal{C}_\lambda, x) = \mathbf{C}$ *then the output of the hardwired circuit* $\mathbf{C}$ *is* $C(x)$.

We can now talk about updatability of hardwired circuits. Note that this captures joint updating of both the circuit as well as the input hardwired into it.

Definition 10 (Updatable Hardwired Circuits)**.** *Consider a family of hardwired circuits* $\{\mathcal{C}[X]_\lambda\}_{\lambda \in \mathbb{N}}$, *where* $\mathcal{C}[X]_\lambda$ *contains* $\mathrm{poly}(\lambda)$*-sized boolean circuits* $C[X] : \perp \rightarrow \{0,1\}^{\ell(\lambda)}$. *Consider a set system of strings* $\mathcal{U} = \{\mathcal{U}_\lambda\}_{\lambda \in \mathbb{N}}$, *where* $\mathcal{U}_\lambda$ *contains a set of strings of length* $\mathrm{poly}(\lambda)$. *We define* $\mathcal{C}[X]$ *to be* $(\mathsf{Upd},\mathcal{U})$**-updatable** *if* $\mathbf{C} \leftarrow \mathsf{Upd}(C[x], \mathbf{u})$, *where* $C[x] \in \mathcal{C}[X]_\lambda, \mathbf{u} \in \mathcal{U}_\lambda$, *then* $\mathbf{C}$ *is also a hardwired circuit.*

We study the notion of updatable randomized encodings with respect to a class of updatable hardwired circuits. In our construction of URE, we restrict our attention to a restricted class of updates, namely, bit-wise updates. We define this notion below.

As in the case of standard circuits, we assume that there is a an efficient binary representation for every hardwired circuit denoted by $\mathsf{binary}(C[x]) \in \{0,1\}^{O(|C|)}$. That is, there is an efficient algorithm that computes $C[x] \mapsto (n, |C|, \mathsf{binary}(C[x]))$, and its inverse.

Definition 11 (Bit-wise Updatable Hardwired Circuits)**.** *A family of hardwired circuits* $\mathcal{C}[X]$ *is* **bit-wise updatable** *if it is* $(\mathsf{Upd}_{\mathsf{bit}},\mathcal{U}_{\mathsf{bit}})$*-updatable, where:*

- $\mathcal{U}_{\mathsf{bit}} = \{(i, b) \in \mathbb{Z}_{\geq 0} \times \{0,1\}\}$.
- $\mathsf{Upd}_{\mathsf{bit}}(C[x], (i, b))$ *takes as input a hardwired circuit* $C[x] \in \mathcal{C}[X]$ *and a update* $(i, b) \in \mathcal{U}_{\mathsf{bit}}$ *indicating the bit index in the circuit representation* $\mathsf{binary}(C[x])$ *to be changed, and a new bit. If* $i > |C|$, *then* $\mathsf{Upd}_{\mathsf{bit}}$ *outputs* $\perp$. *Otherwise, output the circuit that is essentially* $\mathsf{binary}(C[x])$ *except that the* i^{th} *location now contains the bit* b. *Formally, output the circuit* $\mathsf{binary}(C[x]) \oplus 0^{i-1}\ b\ 0^{|\mathsf{binary}(C[x])|-i}$ *which results from changing bit* i *of* $\mathsf{binary}(C[x])$ *to* b.

We now proceed to give a formal definition of URE.

Syntax. A scheme $\mathsf{URE} = (\mathsf{Encode}, \mathsf{GenUpd}, \mathsf{ApplyUpd}, \mathsf{Decode})$ for a $(\mathsf{Upd}, \mathcal{U})$-updatable class of circuits $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda \in \mathbb{N}}$ is defined below. We denote $\mathcal{C}[X]$ to be the corresponding updatable *hardwired* circuit family.

- **Encode,** $(\langle C[x] \rangle_{\mathsf{ure}}, \mathsf{st}) \leftarrow \mathsf{Encode}\left(1^\lambda, C, x\right)$: On input security parameter λ, circuit $C \in \mathcal{C}_\lambda$, input $x \in \{0,1\}^\lambda$, it outputs the joint encoding $\langle C[x] \rangle_{\mathsf{ure}}$ and state st.

- **Generating Secure Update,** $(\langle \mathbf{u} \rangle_{\mathsf{ure}}, \mathsf{st}') \leftarrow \mathsf{GenUpd}(\mathsf{st}, \mathbf{u})$: On input state st, update $\mathbf{u} \in \mathcal{U}_\lambda$, output the secure update $\langle \mathbf{u} \rangle_{\mathsf{ure}}$ along with the new state st'.

- **Apply Secure Update,** $\langle C'[x'] \rangle_{\mathsf{ure}} \leftarrow \mathsf{ApplyUpd}(\langle C[x] \rangle_{\mathsf{ure}}, \langle \mathbf{u} \rangle_{\mathsf{ure}})$: On input randomized encoding $\langle C[x] \rangle_{\mathsf{ure}}$, secure update $\langle \mathbf{u} \rangle_{\mathsf{ure}}$, output the updated randomized encoding $\langle C'[x'] \rangle_{\mathsf{ure}}$.

- **Evaluation,** $\alpha \leftarrow \mathsf{Decode}(\langle C[x] \rangle_{\mathsf{ure}})$: On input randomized encoding $\langle C[x] \rangle_{\mathsf{ure}}$, output the decoded value α.

We associate the above scheme with efficiency, correctness and security properties. We first talk about the efficiency requirement. Modeling of correctness and security properties is tricky and we will deal with them in a separate subsection.

Efficiency. We lay out different efficiency properties associated with the above scheme.

- *Encoding Time*: This property requires that the encoding time of (C, x) is significantly "simpler" than computing $C(x)$. The efficiency aspect can be quantified in many ways – in this work, we define encoding to be efficient if the depth of Encode circuit is smaller than C.

- *Secure Update Generation Time*: This property requires that the runtime of $\mathsf{GenUpd}(\mathsf{st}, \mathbf{u})$ is $p(\lambda, |\mathbf{u}|)$, where p is an a priori fixed polynomial. In other words, the update generation time is independent of the size of the encoded circuit.

- *State Size*: This property requires that the size of the state maintained by the authority is a fixed polynomial in the security parameter. That is, the size of st output by Encode and GenUpd is always $\mathrm{poly}(\lambda)$ independent of the size of the machines and the update sizes.

- *Secure Update Size*: This property states that the size of the secure version of the update should solely depend on the size of the update. Formally, we have the size of the secure update to be $|\langle \mathbf{u} \rangle_{\mathsf{ure}}| = p(\lambda, |\mathbf{u}|)$, where $(\langle \mathbf{u} \rangle_{\mathsf{ure}}, \mathsf{st}') \leftarrow \mathsf{GenUpd}(\mathsf{st}, \mathbf{u})$. Note that any URE scheme that satisfies the above secure update generation time property also satisfies this property.

- *Runtime of Update*: Informally, this property states that the time to update the secure encoding incurs a polynomial overhead in the time to update the plaintext circuit. Formally, the runtime of $\mathsf{ApplyUpd}(\langle C[x] \rangle_{\mathsf{ure}}, \langle \mathbf{u} \rangle_{\mathsf{ure}})$ is $p(\lambda, t, |\mathbf{u}|)$, where t is the time taken to execute $\mathsf{Upd}(C[x], \mathbf{u})$.

Our constructions achieve a restricted version of the above properties. On the positive side, our construction in Section 4 achieves the 'Encoding Time' property and 'Secure Update Generation Time' properties. We use a term to define a URE scheme that satisfies the secure update generation time property – we call it *output compact URE.*

Definition 12 (Output Compact URE). *An URE scheme that is said to be* **output compact** *if it satisfies 'Secure update generation time' property.*

In the case of indistinguishability security, our construction will be output-compact, i.e., the updates will be independent of the output length of the circuit. In the case of simulation-based security, our construction will not achieve output compactness. This is, in fact, inherent and a formal lower bound to this effect can be established along the same lines as in [AGVW13, CIJ+13]. On the flip side, our construction does not satisfy 'Runtime of Update' property.

In Appendix C, we provide a transformation from any URE scheme that satisfies the 'Secure Update Generation Time' property to one that additionally satisfies the 'State Size' property. This transformation uses non-succinct garbled RAMs, and assumes only one-way functions.

3.1 Sequential Updating

We first consider sequential updating process that will be the main focus of this work. For alternate updating processes, refer to Appendix A. Sequential Updating process allows for updating a randomized encoding using multiple patches in a sequential manner. That is, given secure updates $\langle \mathbf{u}_1 \rangle_{\mathsf{ure}}, \ldots, \langle \mathbf{u}_\ell \rangle_{\mathsf{ure}}$, we can update a randomized encoding $\langle C[x] \rangle_{\mathsf{ure}}$ by first applying $\langle \mathbf{u}_1 \rangle_{\mathsf{ure}}$ on $\langle C[x] \rangle_{\mathsf{ure}}$ to obtain the updated encoding $\langle C_1[x_1] \rangle_{\mathsf{ure}}$; next we apply $\langle \mathbf{u}_2 \rangle_{\mathsf{ure}}$ on $\langle C_1[x_1] \rangle_{\mathsf{ure}}$ to obtain the updated encoding $\langle C_2[x_2] \rangle_{\mathsf{ure}}$ and so on. After all the updates, we end up with the updated encoding $\langle C_\ell[x_\ell] \rangle_{\mathsf{ure}}$.

Correctness of Sequential Updating. Intuitively, the correctness property states that computing the randomized encoding $\langle C[x] \rangle_{\mathsf{ure}}$, applying the secure updates $\langle \mathbf{u}_1 \rangle_{\mathsf{ure}}, \ldots, \langle \mathbf{u}_\ell \rangle_{\mathsf{ure}}$ sequentially and finally decoding yields the same result as the output of the circuit obtained by updating the hardwired circuit $C[x]$ by applying the updates $\mathbf{u}_1, \ldots, \mathbf{u}_\ell$ sequentially. We give the formal description below.

Consider a circuit $C \in \mathcal{C}_\lambda$, input $x \in \{0,1\}^\lambda$. Consider a vector of updates $\mathbf{U} \in (\mathcal{U}_\lambda)^q$, where $q(\lambda)$ is a polynomial in λ. Consider the following two processes:

Secure updating process:

1. $(\langle C[x] \rangle_{\mathsf{ure}}, \mathsf{st}_0) \leftarrow \mathsf{Encode}\left(1^\lambda, C, x\right)$.
2. For every $i \in [q]$; $(\langle \mathbf{u}_i \rangle_{\mathsf{ure}}, \mathsf{st}_i) \leftarrow \mathsf{GenUpd}\left(\mathsf{st}_{i-1}, \mathbf{u}_i\right)$, where $\mathbf{u}_i$ is the i^{th} entry in $\mathbf{U}$.
3. Let $\langle C_0[x_0] \rangle_{\mathsf{ure}} := \langle C[x] \rangle_{\mathsf{ure}}$. For every $i \in [q]$; $\langle C_i[x_i] \rangle_{\mathsf{ure}} \leftarrow \mathsf{ApplyUpd}\left(\langle C_{i-1}[x_{i-1}] \rangle_{\mathsf{ure}}, \langle \mathbf{u}_i \rangle_{\mathsf{ure}}\right)$.

Insecure updating process:

1. Let $(C_0, x_0) := (C, x)$. For every $i \in [q]$, we have $C_i[x_i] \leftarrow \mathsf{Upd}(C_{i-1}[x_{i-1}], \mathbf{u}_i)$. The output of $C_q[x_q]$ is $C_q(x_q)$.

We have,

$$\mathsf{Decode}\Big(\langle C_q[x_q] \rangle_{\mathsf{ure}}\Big) = C_q(x_q)$$

Security of Sequential Updating. We consider two different security notions of sequential updatable RE. First, we consider simulation-based notion and then we consider the weaker indistinguishability-based notion.

Our security notions attempt to capture the intuition that an updateable randomized encoding $\langle C_0[x_0] \rangle_{\mathsf{ure}}$ and a sequence of updates $\langle \mathbf{u}_1 \rangle_{\mathsf{ure}}, \ldots, \langle \mathbf{u}_q \rangle_{\mathsf{ure}}$ should reveal only the outputs $C_0(x_0)$, $C_1(x_1)$,... $C_q(x_q)$ where C_i and X_i are defined as in the preceding correctness definition. In addition to hiding the circuits and inputs as in traditional randomized encodings, a URE additionally hides the sequence of updates. Our URE construction satisfies this update-hiding property.

We could instead consider a relaxed notion, in which updates are partially or wholly revealed (modifying the definitions appropriately). Indeed, this is what we will do in the context of updatable garbled circuits (Section 5). In Appendix B, we provide a generic transformation from an update-revealing URE scheme to an update-hiding URE scheme, assuming only the existence of one-way functions.

Simulation-Based Security. We adopt the *real world/ ideal world* paradigm in formalizing the simulation-based security definition of sequential updatable RE. In the real world, the adversary receives a randomized encoding and encodings of updates. All the encodings are generated honestly as per the description of the scheme. In the ideal world, the adversary is provided simulated randomized encodings and encodings of updates. These simulated encodings are generated as a function of the outputs and in particular, the simulation process does not receive as input the circuit, input or the plaintext updates. A sequential updatable RE scheme is secure if an efficient adversary cannot tell apart real world from the ideal world.

The ideal world is formalized by considering a simulator Sim that runs in probabilistic polynomial time. Sim gets as input the output of circuit $C(x)$, the length of C and produces a simulated randomized encoding. We emphasize that Sim does not receive as input C or x. After this, Sim simulates the update encodings. On input length of update $\mathbf{u}_i$, value $C_i(x_i)$, it generates a simulated encoding of $\mathbf{u}_i$. Here, $C_i(x_i)$ is obtained by first updating $C_{i-1}[x_{i-1}]$ using $\mathbf{u}_i$ to obtain $C_i[x_i]$, whose output is $C_i(x_i)$ and also, $C_0[x_0]$ is initialized with $C[x]$. For this discussion, we consider the scenario where the circuit, input along with the updates are fixed at the beginning of the experiment. This is termed as the *selective* setting. We describe the formal experiment in Figure 1.

We present the formal security definition below.

Definition 13 (SIM-secure Sequential URE). *A sequential URE scheme* URE *for* $(\mathsf{Upd},\mathcal{U})$*-updatable class of circuits* $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda\in\mathbb{N}}$ *is said to be* **SIM-secure** *if for every PPT adversary* $\mathcal{A}$*, for every circuit* $C \in \mathcal{C}_\lambda$*, updates* $\mathbf{u}_1, \ldots, \mathbf{u}_q \in \mathcal{U}_\lambda$*, there exists a PPT simulator* Sim *such that the following holds for sufficiently large* $\lambda \in \mathbb{N}$*,*

$$\left|\mathsf{Pr}\left[0 \leftarrow \mathsf{IdealExpt}^{\mathcal{A}}\left(1^\lambda, C, x, \{\mathbf{u}_i\}_{i\in[q]}\right)\right] - \mathsf{Pr}\left[0 \leftarrow \mathsf{RealExpt}^{\mathcal{A}}\left(1^\lambda, C, x, \{\mathbf{u}_i\}_{i\in[q]}\right)\right]\right| \leq \mathsf{negl}(\lambda),$$

where negl *is a negligible function.*

<u>**Experiment** $\mathsf{IdealExpt}^{\mathcal{A}}(1^\lambda, C, x, \{\mathbf{u}_i\}_{i\in[q]})$:</u>

1. $(\langle C[x]\rangle_{\mathsf{ure}}, \mathsf{st}_0) \leftarrow \mathsf{Sim}(1^\lambda, 1^{|C|}, C(x))$.

2. $C_0[x_0] :=$ hardwired circuit of (C, x).

3. $\forall i \in [q]$, $C_i[x_i] \leftarrow \mathsf{Upd}(C_{i-1}[x_{i-1}], \mathbf{u}_i)$.
Let $C_{i-1}(x_{i-1})$ be the output of $C_{i-1}[x_{i-1}]$.

4. $\forall i \in [q]$, $(\langle \mathbf{u}_i\rangle_{\mathsf{ure}}, \mathsf{st}_i) \leftarrow \mathsf{Sim}(\mathsf{st}_{i-1}, 1^{|\mathbf{u}_i|}, C_i(x_i))$.

Output $\mathcal{A}\Big(\langle C[x]\rangle_{\mathsf{ure}}, \langle \mathbf{u}_1\rangle_{\mathsf{ure}}, \ldots, \langle \mathbf{u}_q\rangle_{\mathsf{ure}}\Big)$.

<u>**Experiment** $\mathsf{RealExpt}^{\mathcal{A}}(1^\lambda, C, x, \{\mathbf{u}_i\}_{i\in[q]})$:</u>

1. $(\langle C[x]\rangle_{\mathsf{ure}}, \mathsf{st}_0) \leftarrow \mathsf{Encode}\left(1^\lambda, C, x\right)$.

2. $\forall i \in [q]$, $(\langle \mathbf{u}\rangle_{\mathsf{ure}}, \mathsf{st}_i) \leftarrow \mathsf{GenUpd}\left(\mathsf{st}_{i-1}, \mathbf{u}_i\right)$.

Output $\mathcal{A}\Big(\langle C[x]\rangle_{\mathsf{ure}}, \langle \mathbf{u}_1\rangle_{\mathsf{ure}}, \ldots, \langle \mathbf{u}_q\rangle_{\mathsf{ure}}\Big)$.

Figure 1: Selective Simulation-Based Definition of Sequential URE.

Indistinguishability-Based Security. We formulate a game-based definition between the challenger and the adversary. The adversary makes circuit query (C^0, C^1) along with input x to the challenger. The challenger picks a bit b at random and encodes (C^b, x). This challenge encoding is sent to the adversary. The adversary also queries for secure updates. That is, it sends the pair $(\mathbf{u}_i^0, \mathbf{u}_i^1)$ to the challenger who responds with encoding of $\mathbf{u}_i^b$. The adversary is restricted to "valid" update queries: it should hold that $C_i^0(x_i^0) = C_i^1(x_i^1)$ for every i, where $C_i^0[x_i^0] \leftarrow \mathsf{Upd}(C_{i-1}^0[x_{i-1}], \mathbf{u}_i^0)$ and $C_i^1[x_i^1] \leftarrow \mathsf{Upd}(C_{i-1}^1[x_{i-1}], \mathbf{u}_i^1)$. Furthermore, $C_i^0(x_i^0)$ (resp., $C_i^1(x_i^0)$) is the output of $C_i^0[x_i^0]$ (resp., $C_i^1[x_i^1]$). The update process is initialized by setting $C_0^0[x_0^0] = C^0[x]$ and $C_0^1[x_0^1] = C^1[x]$. If the adversary makes any invalid update queries, the challenger aborts the experiment. In the end, the adversary is required to guess the bit b. We say that the sequential URE scheme is IND-secure if the adversary succeeds with negligible advantage (i.e., with probability negligibly close to 1/2).

We can consider different flavors of IND-based security depending on the order of circuit and update queries made by the adversary. We consider the simplest setting where the adversary is supposed to declare all his queries in the beginning of the game. We call this *selective* setting. We can also consider the *adaptive* setting where the adversary can chose the update and the circuit queries adaptively.

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

- $\mathcal{A}$ sends circuits $(C^0, C^1) \in \mathcal{C}_\lambda$, input $x \in \{0,1\}^\lambda$ to Ch.
- $\mathcal{A}$ then makes a series of update queries. For $i \in [q]$, it sends the i^{th} update query $(\mathbf{u}_i^0, \mathbf{u}_i^1)$ to Ch. Challenger first checks if the following condition holds: letting $C_i^\beta[x_i^\beta] \leftarrow \mathsf{Upd}(C_{i-1}^\beta[x_{i-1}^\beta], \mathbf{u}_i^\beta)$ for $\beta \in \{0,1\}$ and $i \in [q]$:
$$\Big(C^0(x) = C^1(x)\Big) \text{ and } \Big(\forall i \in [q]: \; C_i^0(x_i^0) = C_i^1(x_i^1)\Big).$$
Here, $C_i^0(x_i^0)$ (resp., $C_i^1(x_i^1)$) is the output of $C_i^0[x_i^0]$ (resp., $C_i^1[x_i^1]$). Also, $C_0^0[x_0^0] = C^0[x]$ and $C_0^1[x_0^1] = C^1[x]$. If the condition is not satisfied, Ch aborts the experiment.
- Ch sends $\langle C^b[x] \rangle_{\mathsf{ure}}$ to $\mathcal{A}$, where $(\langle C^b[x] \rangle_{\mathsf{ure}}, \mathsf{st}_0) \leftarrow \mathsf{Encode}(1^\lambda, C^b, x)$, to $\mathcal{A}$. Ch also sends, for every $i \in [q]$, the secure update encoding $(\langle \mathbf{u}_i^b \rangle_{\mathsf{ure}}, \mathsf{st}_i) \leftarrow \mathsf{GenUpd}(\mathsf{st}_{i-1}, \mathbf{u}_i^b)$.
- $\mathcal{A}$ outputs b'.

We state the formal definition of the indistinguishability-based security notion below.

Definition 14 (IND-secure Sequential URE)**.** *A sequential* URE *scheme is* **IND-secure** *if for any PPT adversary* $\mathcal{A}$*, bit* b*, we have* $\mathsf{Pr}[b' = b : b' \leftarrow \mathsf{Expt}_{\mathcal{A}}^{\mathsf{SURE}}(1^\lambda, b)] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$*, for some negligible function* negl.

3.2 IND to SIM-Security

We show how to transform IND-secure URE into SIM-secure URE generically. In the resulting SIM-secure scheme, the size of the updates depends on the output length of the circuit being encoded, even if we start with a IND-secure URE scheme where the size of the updates is independent of the output length of the circuit. That is, suppose (C, x) be the circuit-input pair initially encoded then the size of the secure updates in the resulting SIM-secure scheme depend on the output length of C.

Let $\mathsf{URE}_{\mathsf{IND}} = (\mathsf{Encode}, \mathsf{GenUpd}, \mathsf{ApplyUpd}, \mathsf{Decode})$ be an IND-secure URE scheme. We denote the resulting SIM-secure URE scheme to be $\mathsf{URE}_{\mathsf{SIM}}$. The following transformation follows the same template as that of de Caro et. al. [CIJ+13], which was studied in the context of functional encryption.

- $\mathsf{Encode}\left(1^\lambda, C, x\right)$: On input security parameter λ, circuit $C \in \mathcal{C}_\lambda$, input $x \in \{0,1\}^\lambda$, it computes the encoding of circuit-input pair (C^*, x^*) with respect to $\mathsf{URE}_{\mathsf{IND}}$. That is, it computes the joint encoding w.r.t IND-secure scheme as follows,

$$\left(\langle C^*[x^*]\rangle_{\mathsf{ure}}^{\mathsf{IND}},\ \mathsf{st}_{\mathsf{IND}}\right) \leftarrow \mathsf{URE}_{\mathsf{IND}}.\mathsf{Encode}\left(1^\lambda, C^*, x^*\right)$$

  Here, $x^* = (x, 0, 0)$ and C^* is a circuit that takes input (y, o, b) and outputs $C(y)$ if $b = 0$, else if $b = 1$ it outputs o, else if $b = 2$ it outputs $\perp$.
  Output the encoding $\langle C[x]\rangle_{\mathsf{ure}} = \langle C^*[x^*]\rangle_{\mathsf{ure}}^{\mathsf{IND}}$ and state $\mathsf{st} = (\mathsf{st}_{\mathsf{IND}}, 1^\ell)$, where ℓ is the output length of C.

- $\mathsf{GenUpd}\left(\mathsf{st}, \mathbf{u}\right)$: On input state $\mathsf{st} = \mathsf{st}_{\mathsf{IND}}$, update $\mathbf{u} \in \mathcal{U}_\lambda$, compute the secure updates for $j \in [0, \ell+1]$, where ℓ is the output length of C. In the following, we assign $\mathbf{v}_j = \mathbf{u}$ for every $j \in [\ell]$. We set $\mathbf{v}_0$ as the update that changes the mode b to value 2 and $\mathbf{v}_{\ell+1}$ as the update that changes the mode b to value 0. Looking ahead, in the proof, $\mathbf{v}_j$ would correspond to the j^{th} output bit of the (updated version of) C. For $j \in [0, \ell+1]$;

$$(\langle \mathbf{v}_j\rangle_{\mathsf{ure}}^{\mathsf{IND}}, \mathsf{st}_{\mathsf{IND}}^{j}) \leftarrow \mathsf{URE}_{\mathsf{IND}}.\mathsf{GenUpd}\left(\mathsf{st}_{\mathsf{IND}}^{j-1}, \mathbf{v}_j\right)$$

  In the above, $\mathsf{st}_{\mathsf{IND}}^{-1} = \mathsf{st}_{\mathsf{IND}}$ and denote $\mathsf{st}'_{\mathsf{IND}}$ by $\mathsf{st}^{\ell}_{\mathsf{IND}}$. Output the secure update $\langle \mathbf{u}\rangle_{\mathsf{ure}} = \{\langle \mathbf{v}_j\rangle_{\mathsf{ure}}^{\mathsf{IND}}\}_{j \in [0,\ell+1]}$ and new state $\mathsf{st}' = \mathsf{st}'_{\mathsf{IND}}$.

- $\mathsf{ApplyUpd}\left(\langle C[x]\rangle_{\mathsf{ure}}, \langle \mathbf{u}\rangle_{\mathsf{ure}}\right)$: On input randomized encoding $\langle C[x]\rangle_{\mathsf{ure}}$, secure update $\langle \mathbf{u}\rangle_{\mathsf{ure}} = \{\langle \mathbf{v}_j\rangle_{\mathsf{ure}}^{\mathsf{IND}}\}_{j\in[\ell]}$, compute the following for every $j \in [0, \ell+1]$, where ℓ denotes the output length of C.

$$\langle C_j[x_j]\rangle_{\mathsf{ure}} \leftarrow \mathsf{URE}_{\mathsf{IND}}.\mathsf{ApplyUpd}\left(\langle C_{j-1}[x_{j-1}]\rangle_{\mathsf{ure}}, \langle \mathbf{v}_j\rangle_{\mathsf{ure}}^{\mathsf{IND}}\right)$$

  In the above, $\langle C_{-1}[x_{-1}]\rangle_{\mathsf{ure}} = \langle C[x]\rangle_{\mathsf{ure}}$. Denote $\langle C'[x']\rangle_{\mathsf{ure}} = \langle C_{\ell+1}[x_{\ell+1}]\rangle_{\mathsf{ure}}$.

- $\mathsf{Decode}\left(\langle C[x]\rangle_{\mathsf{ure}}\right)$: Execute the evaluation algorithm Decode of the underlying scheme $\mathsf{URE}_{\mathsf{IND}}$ on input $\langle C[x]\rangle_{\mathsf{ure}}$ and output the result of the evaluation process.

The correctness of the above scheme follows essentially from the correctness of $\mathsf{URE}_{\mathsf{IND}}$: Suppose we have computed an updatable randomized encoding of (C, x). Applying the update $\mathbf{v}_0$ makes the randomized encoding to output $\perp$. Applying the updates $\mathbf{v}_1, \ldots, \mathbf{v}_\ell$ in sequence is equivalent to just applying the update $\mathbf{u}$, where $\mathbf{v}_1 = \cdots = \mathbf{v}_\ell = \mathbf{u}$. Finally applying $\mathbf{v}_{\ell+1}$ results in a randomized encoding of (C', x'), where (C', x') is the circuit-input pair obtained by updating (C, x) with $\mathbf{u}$.

To see why the above scheme is secure, lets take a simple scenario where the adversary only requests for one update. The adversary receives $\langle C[x]\rangle_{\mathsf{ure}} = \langle C^*[x^*]\rangle_{\mathsf{ure}}$ and update encoding $\langle \mathbf{u}\rangle_{\mathsf{ure}} = (\langle \mathbf{v}_0\rangle_{\mathsf{ure}}^{\mathsf{IND}}, \ldots, \langle \mathbf{v}_{\ell+1}\rangle_{\mathsf{ure}}^{\mathsf{IND}})$. If $\langle \mathbf{u}\rangle_{\mathsf{ure}}$ is computed honestly then $\mathbf{v}_j$ is set to be $\mathbf{u}$ for all $j \in [\ell]$. Now, the IND-security of $\mathsf{URE}_{\mathsf{IND}}$ guarantees the following changes to the initial randomized encoding and the secure update encodings are computationally indistinguishable:

- We modify x^* to be $(\perp, o, 0)$, where $o = C(x)$. Recall that x^* was originally $(x, 0, 0)$.

- Let (C', x') be the hardwired circuit obtained by updating (C, x) using $\mathbf{u}$. Let $o' = C'(x')$. We modify $\mathbf{v}_j$, for $j \in [\ell]$ to now update the input x^* from $(\perp,\ o'_1||\cdots||o'_{j-1}||*,\ 2)$ to the new input $(\perp,\ o'_1||\cdots||o'_j||*,\ 2)$. The update $\mathbf{v}_0$ is left intact (i.e., it changes the mode b to 2), while the update $\mathbf{v}_{\ell+1}$ now changes the mode b to value 1.

Note that the circuit C, x and $\mathbf{u}$ are "erased" from the system and hence, the above (modified) encodings can be simulated. The security thus follows.

3.3 On the Necessity of 1-Key Secret Key Compact FE

We show that any output-compact updatable randomized encoding scheme tolerating unbounded number of updates implies 1-key compact functional encryption scheme. We build upon a recent beautiful work of Bitansky et al. [BNPW16] to obtain this result.

We first show how to obtain ϵ-XiO [LPST16b] starting from any output-compact updatable randomized encodings scheme. Once we get XiO, we can apply the transformation of [LPST16b] to obtain 1-key compact public key compact FE. This additionally requires the learning with errors (LWE) assumption. Thus, overall, we conclude that assuming LWE, output-compact updatable randomized encodings imply 1-key public-key compact FE, which also implies 1-key secret-key compact FE.

Below, we start by recalling the notion of XiO and then present our transformation from output-compact updatable randomized encodings to XiO. As stated above, this suffices to obtain our result.

3.3.1 Intermediate Tool: XiO

We recall the definition of XiO introduced in the work of Lin, Pass, Seth and Telang [LPST16b]. Before that, we first recall the definition of indistinguishability obfuscation.

Indistinguishability Obfuscation (iO). The notion of indistinguishability obfuscation (iO), first conceived by Barak et al. [BGI$^+$01], guarantees that the obfuscation of two circuits are computationally indistinguishable as long as they both are equivalent circuits, i.e., the output of both the circuits are the same on every input. Formally,

Definition 15 (Indistinguishability Obfuscator (iO) for Circuits)**.** *A uniform PPT algorithm* iO *is called an indistinguishability obfuscator for a circuit family* $\{\mathcal{C}_\lambda\}_{\lambda\in\mathbb{N}}$*, where* $\mathcal{C}_\lambda$ *consists of circuits* C *of the form* $C : \{0,1\}^n \to \{0,1\}$ *with* $n = n(\lambda)$*, if the following holds:*

- **Completeness:** *For every* $\lambda \in \mathbb{N}$*, every* $C \in \mathcal{C}_\lambda$*, every input* $x \in \{0,1\}^n$*, we have that*

$$\mathsf{Pr}\left[C'(x) = C(x)\ :\ C' \leftarrow \mathsf{iO}(\lambda, C)\right] = 1$$

- **Indistinguishability:** *For any PPT distinguisher* D*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that the following holds: for all sufficiently large* $\lambda \in \mathbb{N}$*, for all pairs of circuits* $C_0, C_1 \in \mathcal{C}_\lambda$ *such that* $C_0(x) = C_1(x)$ *for all inputs* $x \in \{0,1\}^n$ *and* $|C_0| = |C_1|$*, we have:*

$$\Big|\mathsf{Pr}\left[D(\lambda, \mathsf{iO}(\lambda, C_0)) = 1\right] - \mathsf{Pr}[D(\lambda, \mathsf{iO}(\lambda, C_1)) = 1]\Big| \leq \mathsf{negl}(\lambda)$$

- **Polynomial Slowdown***: For every* $\lambda \in \mathbb{N}$*, every* $C \in \mathcal{C}_\lambda$*, we have that* $|\mathsf{iO}(\lambda, C)| = \mathrm{poly}(\lambda, C)$.

Exponentially-Efficient iO (XiO). We recall the notion of XiO defined in [LPST16b]. XiO is an indistinguishability obfuscation with the weaker efficiency requirement that dictates that the size of the obfuscated circuit should be sublinear in the size of the truth table associated with the circuit.

Definition 16. *(Exponentially-Efficient iO (XiO)) For a constant* $\gamma < 1$*, a machine* XiO *is a* γ*-compressing exponentially-efficient indistinguishability obfuscator (XiO) for a circuit class* $\{\mathcal{C}_\lambda\}_{\lambda\in\mathbb{N}}$ *if it satisfies the functionality and indistinguishability in Definition 15 and the following efficiency requirements:*

- **Non-trivial efficiency***: For every* $\lambda \in \mathbb{N}$*, every* $C \in \mathcal{C}_\lambda$*, we have that* $|\mathsf{iO}(\lambda, C)| \leq 2^{n\gamma}\mathrm{poly}(\lambda, C)$*, where* n *is the input length of* C.

3.3.2 Output-Compact URE implies XiO

We now present our transformation from output-compact updatable randomized encodings to XiO. We build upon the ideas used in the recent work of [BNPW16].

Let URE be an output-compact URE scheme. Let c be a constant be such that the size of encoding of (C, x) with respect to URE scheme is $|C|^c \cdot \text{poly}(\lambda)$. We construct an XiO scheme below.

$\mathsf{XiO.Obf}(1^\lambda, C)$: To obfuscate circuit C, compute a randomized encoding $(\langle G[y] \rangle_{\mathsf{ure}}, \mathsf{st}_1) \leftarrow \mathsf{URE.Encode}(1^\lambda, G, y)$, where y is a string of length $\lceil n(1 - \frac{1}{2c}) \rceil$ and initially set $y = 0$. Here, G is defined as follows: G on input y outputs $\{C(i||y)\}_{i \in \left[2^{\lfloor \frac{n}{2c} \rfloor}\right]}$. Then for every $i \in \left[2^{\lceil n(1-\frac{1}{2c}) \rceil}\right]$, compute $(\langle \mathbf{u}_{i+1} \rangle_{\mathsf{ure}}, \mathsf{st}_{i+1} \leftarrow \mathsf{GenUpd}(\mathsf{st}_i, \mathbf{u}_i)$, where $\mathbf{u}_i$ sets $y = i$ in the hardwired circuit $G[y]$. Output the following:

$$C' = (\langle G[y] \rangle_{\mathsf{ure}}, \{\langle \mathbf{u}_i \rangle_{\mathsf{ure}}\}_{i \in [2^?]})$$

$\mathsf{XiO.Eval}(C', x)$: On input x, first compute the truth table of the circuit obfuscated in C' as follows:

- Set $\langle G_0[y_0] \rangle_{\mathsf{ure}} = \langle G[y] \rangle_{\mathsf{ure}}$.
- For every $i \in \left[2^{\lceil n(1-\frac{1}{2c}) \rceil} - 1\right]$, compute $\langle G_{i+1}[y_{i+1}] \rangle_{\mathsf{ure}} \leftarrow \mathsf{ApplyUpd}(\langle G_i[y_i] \rangle_{\mathsf{ure}}, \langle \mathbf{u}_i \rangle_{\mathsf{ure}})$.
- For every $i \in \left[2^{\lceil n(1-\frac{1}{2c} \rceil}\right]$, evaluate $\langle G_i[y_i] \rangle_{\mathsf{ure}}$ to obtain the truth table of C.
- Let the output of the truth table on input x be α.

Output α.

The correctness of the underlying URE scheme implies the correctness of the above XiO scheme. Given any two functionally equivalent circuits C_0 and C_1, we can use the security of the updatable randomized encodings scheme to argue that the obfuscations of C_0 and C_1 are computationally indistinguishable.

We remark about the efficiency of the XiO scheme.

Size of Obfuscated circuit C'. We now remark about the size of the obfuscated circuit. We first calculate the number of updates issued as part of the obfuscated circuit – there are $2^{\lceil n(1-\frac{1}{2c}) \rceil}$ ciphertexts with each one of them of size a fixed polynomial in the security parameter. The size of the randomized encoding of (G, y) is $(2^{\lfloor \frac{n}{2c} \rfloor})^c \cdot \text{poly}(\lambda) = (2^{\lfloor \frac{n}{2} \rfloor}) \cdot \text{poly}(\lambda)$. Thus, total size of the obfuscated circuit is $\max\{2^{\lceil n(1-\frac{1}{2c}) \rceil} \cdot \text{poly}(\lambda),\ (2^{\lfloor \frac{n}{2} \rfloor}) \cdot \text{poly}(\lambda)\} \leq 2^{n^\beta} \cdot \text{poly}(\lambda)$ for some $\beta < 1$.

4 Output-Compact URE from FE

In this section, we present our construction of updatable randomized encodings satisfying output compactness properties.

4.1 Construction

Our goal is to construct an updatable randomized encoding scheme, $\mathsf{URE} = (\mathsf{Encode}, \mathsf{GenUpd}, \mathsf{ApplyUpd}, \mathsf{Decode})$ for $\mathcal{C}$. The main tools we use in our construction are the following. We refer the reader to the preliminaries for the definitions of these primitives.

- Randomized Encoding scheme, $\mathsf{RE} = (\mathsf{RE.Enc}, \mathsf{RE.Dec})$ for the same class of circuits $\mathcal{C}$.
- Compact, Function-private, Single-Key, Secret-key functional encryption (FE) scheme, $\mathsf{FE} = (\mathsf{FE.Setup}, \mathsf{FE.KeyGen}, \mathsf{FE.Enc}, \mathsf{FE.Dec})$.

- Garbling Scheme for circuits, $\mathsf{GC} = (\mathsf{Gen}, \mathsf{GrbCkt}, \mathsf{GrbInp}, \mathsf{EvalGC})$.

We assume, without loss of generality, that all randomized algorithms require only λ-many random bits. We use the above tools to design the algorithms of URE as given below.

The updatable randomized encoding of (C, x) will consist of a (standard) randomized encoding (C, x) and some additional information necessary to carry out the updating process. This additional information consists of a garbled input encoding of C and x with respect to GC, and a FE secret key for a function that takes as input an update and outputs a garbled circuit mapping C and x to a new randomized encoding and new garbled circuit input encodings of C' and x', which are the updated values. Henceforth, we denote by s the size of the representation of the harwired circuit $C[x]$.

$\underline{\mathsf{Encode}\left(1^{\lambda}, C, x\right)}$: On input security parameter λ, perform the following operations.

1. Execute the setup of FE, $\mathsf{FE.MSK} \leftarrow \mathsf{FE.Setup}(1^{\lambda})$.
2. Compute a functional key $\mathsf{FE.SK}_{\mathsf{RRGarbler}} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.MSK}, \mathsf{RRGarbler})$, where $\mathsf{RRGarbler}$ is as defined in Figure 3.
3. In the next step, generate a randomized encoding of input (C, x). That is, compute $\mathsf{RE.Enc}(1^{\lambda}, C, x)$ to obtain $\langle C[x] \rangle_{\mathsf{re}}$.
4. As stated earlier, let s be the size of the representation of $C[x]$. Generate a garbled circuit input encoding of $(C[x], \perp)$ by evaluating $\langle C[x], \perp \rangle_{\mathsf{gc}} \leftarrow \mathsf{GrbInp}(C[x], \perp; r_{\mathsf{gc}})$, where r_{gc} is the randomness used to garble the input. Here we view $(C[x], \perp)$ as an *input* (to the circuit $\mathsf{RelockRelease}$).
5. Output as the randomized encoding the tuple,
$$\langle C[x] \rangle_{\mathsf{ure}} = \Big(\mathsf{FE.SK}_{\mathsf{RRGarbler}},\ \langle C[x] \rangle_{\mathsf{re}},\ \langle C[x], \perp \rangle_{\mathsf{gc}} \Big)$$
and set the state to be $\mathsf{st} = (\mathsf{FE.MSK},\ r_{\mathsf{gc}})$.

$\underline{\mathsf{GenUpd}\left(\mathsf{st}_i,\ \mathbf{u}_{i+1}\right)}$: It takes as input the state $\mathsf{st}_i = (\mathsf{FE.MSK}, r_{\mathsf{gc},i})$ and update $\mathbf{u}_{i+1}$.

1. Sample random coins $r_{\mathsf{re},i+1}$ and $r_{\mathsf{gc},i+1}$. Let $\mathsf{mode} = 0$.
2. Generate the FE ciphertext,
$$\mathsf{CT}_{i+1} \leftarrow \mathsf{FE.Enc}\left(\mathsf{FE.MSK},\ (\mathbf{u}_{i+1},\ \perp,\ r_{\mathsf{gc},i},\ r_{\mathsf{gc},i+1},\ r_{\mathsf{re},i+1},\ \mathsf{mode})\right)$$
3. Set the new state $\mathsf{st}_{i+1} = (\mathsf{FE.MSK},\ r_{\mathsf{gc},i+1})$.
4. Output $\langle \mathbf{u}_{i+1} \rangle_{\mathsf{ure}} = \mathsf{CT}_{i+1}$.

$\underline{\mathsf{ApplyUpd}\left(\langle C_i[x_i] \rangle_{\mathsf{ure}}, \langle \mathbf{u}_{i+1} \rangle_{\mathsf{ure}}\right)}$: On input circuit encoding $\langle C_i[x_i] \rangle_{\mathsf{ure}}$ and update encoding $\langle \mathbf{u}_{i+1} \rangle_{\mathsf{ure}} = \mathsf{CT}_{i+1}$, execute the following.

1. Parse the circuit encoding as:
$$\langle C_i[x_i] \rangle_{\mathsf{ure}} = \Big(\mathsf{FE.SK}_{\mathsf{RRGarbler}},\ \langle C_i[x_i] \rangle_{\mathsf{re}},\ \langle C_i[x_i], \perp \rangle_{\mathsf{gc}} \Big)$$
2. Execute the FE decryption, $\mathsf{FE.Dec}(\mathsf{FE.SK}_{\mathsf{RRGarbler}}, \mathsf{CT}_{i+1})$ to obtain $\langle \mathsf{RelockRelease}_{i+1} \rangle_{\mathsf{gc}}$.

$\underline{\mathsf{RelockRelease}_{i+1}}$

Input: $C_i^0[x_i^0]$, $C_i^1[x_i^1]$
Hard-coded values: $\mathbf{u}_{i+1}^0$, $\mathbf{u}_{i+1}^1$, $r_{\mathsf{gc},i+1}$, $r_{\mathsf{re},i+1}$, and mode

- Update both the hardwired circuits $C_i^b[x_i^b]$ using $\mathbf{u}_{i+1}^b$:

$$C_{i+1}^b[x_{i+1}^b] \leftarrow \mathsf{Upd}(C_i^b[x_i^b], \mathbf{u}_{i+1}^b)$$

- Encode the updated hardwired circuit $C_{i+1}^{\mathsf{mode}}[x_{i+1}^{\mathsf{mode}}$:

$$\langle C_{i+1}^{\mathsf{mode}}[x_{i+1}^{\mathsf{mode}}]\rangle_{\mathsf{re}} \leftarrow \mathsf{RE.Enc}\Big(C_{i+1}^{\mathsf{mode}}[x_{i+1}^{\mathsf{mode}}];\ r_{\mathsf{re},i+1}\Big)$$

- Compute the randomized encoding of the input $\big(C_{i+1}^0[x_{i+1}^0],\ C_{i+1}^1[x_{i+1}^1]\big)$:

$$\langle C_{i+1}^0[x_{i+1}^0],\ C_{i+1}^1[x_{i+1}^1]\rangle_{\mathsf{gc}} \leftarrow \mathsf{GrbInp}\Big(\ \big(C_{i+1}^0[x_{i+1}^0], C_{i+1}^1[x_{i+1}^1]\big)\ ;\ r_{\mathsf{gc},i+1}\Big)$$

- Output $\Big(\langle C_{i+1}^{\mathsf{mode}}[x_{i+1}^{\mathsf{mode}}]\rangle_{\mathsf{re}},\ \langle C_{i+1}^0[x_{i+1}^0], C_{i+1}^1[x_{i+1}^1]\rangle_{\mathsf{gc}}\Big)$

Figure 2:

$\underline{\mathsf{RRGarbler}}$

Input: $(\mathbf{u}_{i+1}^0,\ \mathbf{u}_{i+1}^1,\ r_{\mathsf{gc},i},\ r_{\mathsf{gc},i+1},\ r_{\mathsf{re},i+1},\ \mathsf{mode})$

Compute the garbled circuit encoding of $\mathsf{RelockRelease}_{i+1}$*, which is defined in Figure 2:*

$$\langle \mathsf{RelockRelease}_{i+1}\rangle_{\mathsf{gc}} \leftarrow \mathsf{GrbCkt}\Big(\mathsf{RelockRelease}_{i+1};\ r_{\mathsf{gc},i}\Big)$$

Output $\langle \mathsf{RelockRelease}_{i+i}\rangle_{\mathsf{gc}}$.

Figure 3:

3. Execute the decode algorithm of the garbling scheme,

$$(\langle C_{i+1}[x_{i+1}]\rangle_{\mathsf{re}}, \langle C_{i+1}[x_{i+1}], \perp\rangle_{\mathsf{gc}}) \leftarrow \mathsf{EvalGC}(\langle \mathsf{RelockRelease}_{i+1}\rangle_{\mathsf{gc}}, \langle C_i[x_i]\rangle_{\mathsf{gc}})$$

   That is, the decode algorithm outputs the randomized encoding of updated hardwired circuit $C_{i+1}[x_{i+1}]$ and also wire keys of $(C_{i+1}[x_{i+1}], \perp)$ that will be input to the next level garbled circuit.

4. Output $\Big(\mathsf{FE.SK}_{\mathsf{RRGarbler}}, \langle C_{i+1}[x_{i+1}]\rangle_{\mathsf{re}}, \langle C_{i+1}[x_{i+1}], \perp\rangle_{\mathsf{gc}}\Big)$.

$\underline{\mathsf{Decode}\ (\langle C_i[x_i]\rangle_{\mathsf{ure}})}$: On input encoding $\langle C_i[x_i]\rangle_{\mathsf{ure}} = (\mathsf{FE.SK}_{\mathsf{RRGarbler}},\ \langle C_i[x_i]\rangle_{\mathsf{re}},\ \langle C_i[x_i], \perp\rangle_{\mathsf{gc}})$, decode the encoding $\langle C_i[x_i]\rangle_{\mathsf{re}}$ by executing $\mathsf{RE.Dec}(\langle C_i[x_i]\rangle_{\mathsf{re}})$ to obtain α. Output the value α.

We first prove that the above construction satisfies the correctness and efficiency properties.

Correctness. Consider a circuit $C \in \mathcal{C}_\lambda$, input $x \in \{0,1\}^\lambda$. Consider a vector of updates $\mathbf{U} \in (\mathcal{U}_\lambda)^q$, where $q(\lambda)$ is a polynomial in λ. Consider the following two processes:
Secure updating process:

1. $(\langle C[x]\rangle_{\mathsf{ure}}, \mathsf{st}_0) \leftarrow \mathsf{Encode}\left(1^\lambda, C, x\right)$.
2. For every $i \in [q]$; $(\langle \mathbf{u}_i\rangle_{\mathsf{ure}}, \mathsf{st}_i) \leftarrow \mathsf{GenUpd}\left(\mathsf{st}_{i-1}, \mathbf{u}_i\right)$, where $\mathbf{u}_i$ is the i^{th} entry in $\mathbf{U}$.
3. Let $\langle C_0[x_0]\rangle_{\mathsf{ure}} := \langle C[x]\rangle_{\mathsf{ure}}$. For every $i \in [q]$; $\langle C_i[x_i]\rangle_{\mathsf{ure}} \leftarrow \mathsf{ApplyUpd}\left(\langle C_{i-1}[x_{i-1}]\rangle_{\mathsf{ure}}, \langle \mathbf{u}_i\rangle_{\mathsf{ure}}\right)$.

Insecure updating process:

1. Let $(C_0, x_0) := (C, x)$. For every $i \in [q]$, we have $C_i[x_i] \leftarrow \mathsf{Upd}(C_{i-1}[x_{i-1}], \mathbf{u}_i)$. The output of $C_q[x_q]$ is $C_q(x_q)$.

We need to show,

Theorem 6. $\mathsf{Decode}\left(\langle C_q[x_q]\rangle_{\mathsf{ure}}\right) = C_q(x_q)$

Proof. We first define the following notation: We say that $\langle G\rangle_{\mathsf{gc}}$ is a valid garbled circuit of G if there exists randomness r such that $\langle G\rangle_{\mathsf{gc}} \leftarrow \mathsf{GrbCkt}(G;\ r)$. Further, we say that $\langle z\rangle_{\mathsf{gc}}$ is a valid garbled input encoding of z if there is randomness r' such that $\langle z\rangle_{\mathsf{gc}} \leftarrow \mathsf{GrbInp}(z; r')$. Consider the following claim.

Claim 1. *For every $i \in [q]$, the output of FE decryption,* $\mathsf{FE.Dec}(\mathsf{FE.SK}_{\mathsf{RRGarbler}}, \mathsf{CT}_{i+1})$*;* $\langle \mathsf{RelockRelease}_{i+1}\rangle_{\mathsf{gc}}$ *is a valid garbled circuit of* $\mathsf{RelockRelease}_{i+1}$.

The proof of the above claim follows from the correctness of FE.

Claim 2. *The output of evaluation of garbling scheme,*

$$\left(\langle C_{i+1}[x_{i+1}]\rangle_{\mathsf{re}}, \langle C_{i+1}[x_{i+1}], \perp\rangle_{\mathsf{gc}}\right) \leftarrow \mathsf{EvalGC}\left(\langle \mathsf{RelockRelease}_{i+1}\rangle_{\mathsf{gc}}, \langle C_i[x_i]\rangle_{\mathsf{gc}}\right)$$

yields a randomized encoding of $C_{i+1}[x_{i+1}]$ and a valid garbled input encoding $\langle C_{i+1}[x_{i+1}], \perp\rangle_{\mathsf{gc}}$ of $C_{i+1}[x_{i+1}]$.

Proof. We prove this by induction. Initially, the user is given a valid input encoding of $(C_0[x_0], \perp)$, i.e., $\langle C_0[x_0], \perp\rangle_{\mathsf{gc}}$. From Claim 1, the output of $\mathsf{FE.Dec}(\mathsf{FE.SK}_{\mathsf{RRGarbler}}, \mathsf{CT}_1)$ is a valid garbled circuit $\langle \mathsf{RelockRelease}_1\rangle_{\mathsf{gc}}$ of $\mathsf{RelockRelease}_1$ (this circuit corresponds to the first update). From the correctness of garbling schemes, it follows that the evaluation of $\langle \mathsf{RelockRelease}_1\rangle_{\mathsf{gc}}$ on the input encoding $\langle C_0[x_0], \perp\rangle_{\mathsf{gc}}$ is a valid garbled input encoding $\langle C_1[x_1], \perp\rangle_{\mathsf{gc}}$ and a randomized encoding of $C_1[x_1]$. This proves the base case.

Suppose the statement is true for some $i \in [q]$. From Claim 1, it follows that the output of $\mathsf{FE.Dec}(\mathsf{FE.SK}_{\mathsf{RRGarbler}}, \mathsf{CT}_{i+1})$ is a valid garbled circuit $\langle \mathsf{RelockRelease}_{i+1}\rangle_{\mathsf{gc}}$ of $\mathsf{RelockRelease}_{i+1}$. From the correctness of garbling schemes, it follows that the output of evaluation of $\langle \mathsf{RelockRelease}_{i+1}\rangle_{\mathsf{gc}}$ on the input encoding $\langle C_i[x_i], \perp\rangle_{\mathsf{gc}}$) is a valid garbled input encoding $\langle C_{i+1}[x_{i+1}], \perp\rangle_{\mathsf{gc}}$ and a randomized encoding of $C_{i+1}[x_{i+1}]$. This proves the claim. □

From the above claim, we have that the output of $\mathsf{Decode}\left(\langle C_q[x_q]\rangle_{\mathsf{ure}}\right)$ is $C_q(x_q)$.

□

Efficiency. We show that the above scheme satisfies the following properties:

1. *State Size*: The size of st output by Encode is a fixed polynomial in λ.
2. *Secure Update Generation Time*: The time to generate a secure update $\mathbf{u}$ takes time polynomial in $(\lambda, |\mathbf{u}|)$. This follows from the compactness property of the underlying FE scheme – this guarantees that the running time of an encryption of m is a fixed polynomial in $(|m|, \lambda)$. Note that this property also shows that that the size of the secure update is a fixed polynomial in $(\lambda, |\mathbf{u}|)$.

Since the URE scheme satisfies both the encoding time and the secure update generation time properties, it is *output compact* (Definition 12).

The algorithm Encode can be implemented in NC^1 if we additionally assume: (i) the FE scheme we employ has key generation in NC^1 and, (ii) RE has an encode algorithm in NC^1 and, (iii) GC has garbled input encoding algorithm in NC^1. We can instantiate (ii) and (iii) under standard cryptographic assumptions such as decisional Diffie-Helman and learning with errors. We can base (i) on the existence of a function-private *collusion-resistant* secret key FE. This follows from the works of [AJ15, BV15a, AJS15a].

4.2 Proof of Security

We prove the security of URE with respect to indistinguishability-based security definition. Let C_0^0, C_0^1, x be the initial circuits and input submitted by the adversary in the security proof. And let, $(\mathbf{u}_1^0, \mathbf{u}_1^1), \ldots, (\mathbf{u}_q^0, \mathbf{u}_q^1)$ be the tuple of updates. There are two "chains" of updating processes with the 0^{th} chain starting from C_0^0 and 1^{st} chain starting from C_1^1. The i^{th} "bead" on 0^{th} (resp., 1^{st}) chain corresponds to update $\mathbf{u}_i^0$ (resp., $\mathbf{u}_i^1$).

In the security proof, we start with the real experiment where challenge bit 0 is used. That is, the 0^{th} chain is active in the experiment. In the next step, we introduce the 1^{st} chain, along with the already present 0^{th}, into the experiment. However even in this step, 0^{th} chain is still active – that is, generating the randomized encoding at every step is performed using the 0^{th} chain. In the next intemediate hybrids, we slowly switch from 0^{th} chain being activated to 1^{st} chain being activated. In the i^{th} intermediate step, the first i beads on 1^{st} chain are active and on the 0^{th} chain, all except the first i beads are active – this means that the first i updated randomized encodings are computed using the 1^{st} chain and the rest of them are computed using 0^{th} chain. At the end of these intermediate hybrids, we have the 1^{st} chain to be active and 0^{th} chain to be completely inactive. At this stage, we can remove the 0^{th} chain and this completes the proof.

The two chains described above are implemented in a sequence of garbled circuits, that we call *cascaded* garbled circuits. That is, every i^{th} garbled circuit in this sequence produces wire keys for the next garbled circuit. Every garbled circuit in this sequence is a result of ApplyUpd procedure and encapsulates, for some i, the i^{th} beads on both the chains. In order to move from the i^{th} intermediate step to $(i+1)^{th}$ intermediate step, we use the security of garbled circuits. But since these garbled circuits are not given directly, but instead produced by a FE key, we need to make use of security of FE to make this switch work. With this high level proof overview, we present the formal proof below.

Before we present the hybrids, we first introduce two lemmas relevant to the security of FE and garbled circuits that will be useful later.

FE Distributional lemma. In the security proof, we will make frequent use of the following lemma (which is implicit in many previous works). In the context of secret key, function private, semantically secure functional encryption, the lemma gives us a way to switch from a secret key of some function f_0 and an encryption of a randomized message $M^{(0)}$ to a secret key of a different function f_1 and encryption of a different randomized message $M^{(1)}$, even when $f_0(M^{(0)})$ may not equal $f_1(M^{(1)})$. Whereas the security definition of (function private) functional encryption only makes explicit guarantees, we leverage the indistinguishability of the distributions $f_0(M^{(0)})$ and $f_1(M^{(1)})$, where the randomness is taken over the choice of the message, to switch from one to the other.

We state the lemma in the presence of some auxiliary information about $M^{(0)}$ and $M^{(1)}$, along with additional FE-ciphertexts for (randomized) messages $M_2, \ldots, M_q$. We first introduce some notation.

Definition 17 (Admissible Functions)**.** *A function family $\mathcal{F}$ is said to be* admissible *if $f \in \mathcal{F}_\lambda$ and y is in the co-domain of f, then the hard-coded function f^y is also contained in $\mathcal{F}_\lambda$, where*

$$f^y(m) = \begin{cases} y & \text{if } m = \bot \\ f(m) & \text{otherwise.} \end{cases}$$

Armed with the above definition, we state the FE distributional lemma next.

Lemma 1 (FE Distributional Lemma)**.** *Let $\mathsf{FE} = (\mathsf{FE.Setup}, \mathsf{FE.KeyGen}, \mathsf{FE.Enc}, \mathsf{FE.Dec})$ be a 1-collusion, function-private, selectively secure functional encryption scheme over a admissible function space (Definition 17) $\mathcal{F} = \{\mathcal{F}_\lambda\}_{\lambda \in \mathbb{N}}$ and a message space $\mathcal{M} = \{\mathcal{M}_\lambda\}_{\lambda \in \mathbb{N}}$.*

- *For each $\lambda \in \mathbb{N}$, let $M_1^{(0)}$, $M_1^{(1)}$, and $M_2, \dots, M_q$ be random variables over $\mathcal{M}_\lambda$, where joint distribution is publicly known.*
- *Let $(f_0, f_1) \in \mathcal{F}_\lambda$ be public functions such that $\Pr[f_0(M_i) = f_1(M_i)] = 1$ for all $i \geq 2$.*
- *For $b \in \{0,1\}$, let Aux_b be a random variable over $\{0,1\}^*$ drawn from some joint distribution with $(M_1^{(b)}, M_2, \dots, M_q)$.*

Suppose that $\mathcal{M}_\lambda$ contains a special message $\bot$ that is not in the support of any of the random variables M[3]*. Then it holds that: If*

$$\left(f_0(M_1^{(0)}), \mathrm{Aux}_0\right) \approx_c \left(f_1(M_1^{(1)}), \mathrm{Aux}_1\right)$$

then:

$$\begin{aligned} &\left(\mathsf{FE.SK}_0, \mathsf{FE.CT}_1^{(0)}, \mathsf{FE.CT}_2, \dots, \mathsf{FE.CT}_q, \mathrm{Aux}_0\right) \\ \approx_c &\left(\mathsf{FE.SK}_1, \mathsf{FE.CT}_1^{(1)}, \mathsf{FE.CT}_2, \dots, \mathsf{FE.CT}_q, \mathrm{Aux}_1\right) \end{aligned}$$

where the probabilities are taken over the choice of $M_1^{(0)}$, $M_1^{(1)}$, $M_2, \dots, M_q$, and the following probability experiment:

$$\begin{aligned} &\mathsf{FE.MSK} \leftarrow \mathsf{FE.Setup}(1^\lambda) \\ &\mathsf{FE.SK}_b \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.MSK}, f_b) \quad \forall b \in \{0,1\} \\ &\mathsf{FE.CT}_1^{(b)} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.MSK}, M_1^{(b)}) \quad \forall b \in \{0,1\} \\ &\mathsf{FE.CT}_i \leftarrow \mathsf{FE.Enc}(\mathsf{FE.MSK}, M_i) \quad \forall i \in [2, q] \end{aligned}$$

Proof. The proof of the lemma proceeds by first hardwiring the output of the function $f_0(M_1^{(0)})$ in the functional key $\mathsf{FE.SK}_0$. Once this is done, the message in $\mathsf{FE.CT}_0$ can be switched to $\bot$ since this does not change the output of the function. After this, the hardwiring can be switched from $f_0(M_1^{(0)})$ to $f_1(M_1^{(1)})$ (along with auxiliary information). The next steps mimics the above steps in a backward fashion. We present the hybrids below. In the hybrids, we just describe the distributions received by the adversary.

Hybrid $\mathbf{H}_0$: $\left(\mathsf{FE.SK}_0, \mathsf{FE.CT}_1^{(0)}, \mathsf{FE.CT}_2, \dots, \mathsf{FE.CT}_q, \mathrm{Aux}_0\right)$

[3]This requirement along with requirement that the scheme be defined for admissible functions will be used in the proof of the lemma. Existing work in functional encryption (and obfuscation) that use similar arguments often make explicit use of "padding" – arbitrarily increasing the description size of the underlying function to allow for hardcoding. Exactly this type of padding construction can be used to imbue FE with this "closure under hardcoding" property, but we find it conceptually simpler to state the requirement explicitly.

Hybrid $\mathbf{H}_1$: $\Big(\mathsf{FE.SK}_0^{Y_0}, \mathsf{FE.CT}_1^{(0)}, \mathsf{FE.CT}_2, \ldots, \mathsf{FE.CT}_q, \mathrm{Aux}_0\Big)$ where $Y_0 = f_0(M_1^{(0)})$ and $\mathsf{FE.SK}_0^{Y_0} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.MSK}, f_0^{Y_0})$.

Indistinguishability from $\mathbf{H}_0$ follows from the function-privacy of FE, because the functionalities agree on all encrypted inputs.

Hybrid $\mathbf{H}_2$: $\Big(\mathsf{FE.SK}_0^{Y_0}, \mathsf{FE.CT}_1^{(\perp)}, \mathsf{FE.CT}_2, \ldots, \mathsf{FE.CT}_q, \mathrm{Aux}_0\Big)$ where $\mathsf{FE.CT}_1^{(\perp)} \leftarrow \mathsf{FE.Enc}(\mathsf{FE.MSK}, \perp)$.

Indistinguishability follows from the selective security of FE, as $f_0^{M_1^{(0)}}$ agrees on $M_1^{(0)}$ and $\perp$.

Hybrid $\mathbf{H}_3$: $\Big(\mathsf{FE.SK}_0^{Y_1}, \mathsf{FE.CT}_1^{(\perp)}, \mathsf{FE.CT}_2, \ldots, \mathsf{FE.CT}_q, \mathrm{Aux}_1\Big)$ where $Y_1 = f_1(M_1^{(1)})$ and $\mathsf{FE.SK}_0^{Y_1} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.MSK}, f_0^{Y_1})$.

This follows from the indistinguishability of (Y_0, Aux_0) and (Y_1, Aux_1).

Hybrid $\mathbf{H}_4$: $\Big(\mathsf{FE.SK}_1^{Y_1}, \mathsf{FE.CT}_1^{(\perp)}, \mathsf{FE.CT}_2, \ldots, \mathsf{FE.CT}_q, \mathrm{Aux}_1\Big)$ where $\mathsf{FE.SK}_1^{Y_1} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.MSK}, f_1^{Y_1})$.

Indistinguishability from $\mathbf{H}_3$ follows from the function-privacy of FE, because the functionalities agree on all encrypted inputs.

Hybrid $\mathbf{H}_5$: $\Big(\mathsf{FE.SK}_1, \mathsf{FE.CT}_1^{(1)}, \mathsf{FE.CT}_2, \ldots, \mathsf{FE.CT}_q, \mathrm{Aux}_1\Big)$ by an argument mirroring the indistinguishability of $\mathbf{H}_0$ and $\mathbf{H}_2$.
This completes the proof. □

Cascaded Garbled Circuits We start by explaining the notion of *cascaded circuits.* It is a sequence of circuits, initialized with an input, where the i^{th} circuit in the sequence produces input to the $(i+1)^{th}$ circuit. Every circuit also produces additional output that will not be part of the input to the next circuit in the sequence. We define this notion formally below.

Definition 18 (Cascaded Circuits)**.** *Let $C_1,\ldots,C_q$ be circuits mapping $\{0,1\}^n$ to $\{0,1\}^m$ for some $m > n$. We parse the output as $C_i(x_i) = x_{i+1}\|y_i$ where $x_{i+1} \in \{0,1\}^n$ and $y_i \in \{0,1\}^{m-n}$. We say that $(x_1, C_1, \ldots, C_q)$ are* **cascaded circuits** *if they are evaluated as follows:*

$$\begin{aligned} x_2, y_1 &\leftarrow C_1(x_1) \\ x_3, y_2 &\leftarrow C_2(x_2) \\ &\vdots \\ x_{q+1}, y_q &\leftarrow C_q(x_q) \end{aligned}$$

The output of cascaded circuits is defined to be $(y_1, \ldots, y_q)$.

Using the above notion, we can define the concept of cascaded garbled circuits. It is a sequence of garbled circuits, where the i^{th} garbled circuit in the sequence produces wire keys to the $(i+1)^{th}$ garbled circuit. As before, every garbled circuit also produces additional output.

Definition 19 (Cascaded Garbled Circuits)**.** *Let $\mathsf{GC} = (\mathsf{Gen}, \mathsf{GrbCkt}, \mathsf{GrbInp}, \mathsf{EvalGC})$ be a circuit garbling scheme. Let $C_1,\ldots,C_q$ be circuits mapping $\{0,1\}^n$ to $\{0,1\}^m$ for some $m > n$. For a circuit C_i and a garbling key $r \leftarrow \mathsf{Gen}(1^\lambda)$, define the circuit $G[C_i, r]$ on input x_i to be:*

$$G[C_i, r](x_i) = \mathsf{GrbInp}(r, x_{i+1})\|y_i.$$

For the cascaded circuits ($x_1 \in \{0,1\}^n$, $C_1, \ldots, C_q$), we define ***cascaded garbled circuits*** *to be*

$$\left(\langle x_1\rangle_{\mathsf{gc}}, \langle G_1\rangle_{\mathsf{gc}}, \langle G_2\rangle_{\mathsf{gc}}, \ldots, \langle G_q\rangle_{\mathsf{gc}}\right) \text{ sampled according to}$$
$$(\mathsf{GrbInp}(r_1, x_1), \mathsf{GrbCkt}(r_1, G_1), \mathsf{GrbCkt}(r_2, G_2), \ldots, \mathsf{GrbCkt}(r_q, G_q))$$

where $r_i \leftarrow \mathsf{Gen}(1^\lambda)$ and $G_i = G[C_i, r_{i+1}]$. The output of cascaded garbled circuits is defined to be $(y_1, \ldots, y_q)$.

Each G_i outputs a garbled input for the next garbled circuit $\langle G_{i+1}\rangle_{\mathsf{gc}}$, along with some additional output y_i. Namely, the cascaded garbled circuit is evaluated analogously to the cascaded circuits:

$$\begin{aligned}
\langle x_2\rangle_{\mathsf{gc}},\ y_1 &\leftarrow \mathsf{EvalGC}(\langle G_1\rangle_{\mathsf{gc}}, \langle x_1\rangle_{\mathsf{gc}})\\
\langle x_3\rangle_{\mathsf{gc}},\ y_2 &\leftarrow \mathsf{EvalGC}(\langle G_2\rangle_{\mathsf{gc}}, \langle x_2\rangle_{\mathsf{gc}})\\
&\vdots\\
\langle x_{q+1}\rangle_{\mathsf{gc}}, y_q &\leftarrow \mathsf{EvalGC}(\langle C_q\rangle_{\mathsf{gc}}, \langle x_q\rangle_{\mathsf{gc}})
\end{aligned}$$

and the output is defined to be $(y_1, \ldots, y_q)$.

We are now ready to state a lemma about the security of cascaded garbled circuits. This lemma implicitly in several different contexts such as secure computation on the web [HLP11, GHV10], garbled RAMS [LO13, GHL+14, GLOS15, GLO15], indistinguishability obfuscation for Turing machines [BGL+15] and adaptive garbled circuits [HJO+15].

Lemma 2 (SIM-security of Cascaded Garbled Circuits)**.** *Suppose that* GC *is a SIM-secure garbled circuits scheme (Definition 2). Let* SimGC *be the PPT simulator associated with* GC*. Then there exists a PPT simulator* Sim *such that for all cascaded circuits $(x_1, C_1, \ldots, C_q)$ with output $(y_1, \ldots, y_q)$, the distribution over the cascaded garbled circuit $\left(\langle x_1\rangle_{\mathsf{gc}}, \langle G_1\rangle_{\mathsf{gc}}, \langle G_2\rangle_{\mathsf{gc}}, \ldots, \langle G_{q-1}\rangle_{\mathsf{gc}}, \langle G_q\rangle_{\mathsf{gc}}\right)$ is computationally indistinguishable from the output distribution of* $\mathsf{Sim}\left(1^\lambda, \phi(C_1), \ldots, \phi(C_q), y_1, \ldots, y_q\right)$*, where $\phi(C_i)$ denotes the topology of C_i.*

Proof. We define the simulator Sim using the simulator of garbled circuits, $\mathsf{Sim}_{\mathsf{gc}}$. The simulator Sim takes as input $(y_1, \ldots, y_q)$ and does the following:

- Let ℓ be the output length of G_q. Let $(x^{\mathsf{sim}}_{q+1}, G^{\mathsf{sim}}_{q+1}) \leftarrow \mathsf{Sim}_{\mathsf{gc}}(0^\ell)$.
- For i from q to 1, let $(x^{\mathsf{sim}}_i, G^{\mathsf{sim}}_i) \leftarrow \mathsf{Sim}_{\mathsf{gc}}(x^{\mathsf{sim}}_{i+1}, y_i)$.
- Output $\left(x^{\mathsf{sim}}_1, G^{\mathsf{sim}}_1, \ldots, G^{\mathsf{sim}}_q\right)$.

The proof of computational indistinguishability proceeds by a sequence of hybrids. The initial hybrid is the real cascaded garbled circuit, and the final hybrid is the simulated cascaded garbled circuit. The intermediate hybrids are defined by simulating the garbling of x_1, along with the first i circuits $G_1, \ldots, G_i$, while using GC to garbled the remaining circuits $G_{i+1}, \ldots, G_q$. The final reduction uses the SIM security of GC to establish that each adjacent pair of hybrids are indistinguishable. We briefly describe the hybrids below.

Hybrid $\mathbf{H}_1$: $\left(\langle x_1\rangle_{\mathsf{gc}}, \langle G_1\rangle_{\mathsf{gc}}, \langle G_2\rangle_{\mathsf{gc}}, \ldots, \langle G_{q-1}\rangle_{\mathsf{gc}}, \langle G_q\rangle_{\mathsf{gc}}\right)$ - All the garbled circuits are computed according to GC.

Hybrid $\mathbf{H}_{2.i}$: $\left(x^{\mathsf{sim}}_1, G^{\mathsf{sim}}_1, G^{\mathsf{sim}}_2, \ldots, G^{\mathsf{sim}}_i, \langle G_{i+1}\rangle_{\mathsf{gc}}, \ldots, \langle G_{q-1}\rangle_{\mathsf{gc}}, \langle G_q\rangle_{\mathsf{gc}}\right)$ - the first i garbled circuits are simulated and the rest are computed according to GC.

$\mathbf{H}_{2.0}$ is identical to $\mathbf{H}_1$. The indistinguishability of $\mathbf{H}_{2.i}$ and $\mathbf{H}_{2.i+1}$ follows from the security of garbled circuits.

Hybrid $\mathbf{H}_3$: $\left(x_1^{\mathsf{sim}}, G_1^{\mathsf{sim}}, \ldots, G_q^{\mathsf{sim}}\right)$ - All the garbled circuits are simulated.

The hybrid $\mathbf{H}_{2.q+1}$ is identical to $\mathbf{H}_3$.

□

Main hybrids. We prove that the above is a IND-secure Sequential URE scheme. We now present the main sequence of hybrids, and only then demonstrate their indistinguishability. Let q be the number of update queries $(\mathbf{u}_i^0, \mathbf{u}_i^1)$ the adversary makes in the IND security game in Definition 14.

The first hybrid corresponds to the IND-experiment $\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SURE}}(1^\lambda, 0)$ given in Definition 14. Then we move to a hybrid in which the security experiment is run using modified versions of Encode and GenUpd, with outputs corresponding to updatable randomized encoding of circuit C^0, input x^0, and updates $\mathbf{u}_i^0$.

We then proceed to change the encodings one-by-one until all all the encodings correspond to circuit C^1, input x^1, and updates $\mathbf{u}_i^1$. Finally we revert to the unmodified versions of Encode and GenUpd, yielding an experiment corresponding to the IND-experiment $\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SURE}}(1^\lambda, 1)$.

$\mathbf{H}^0_{\text{real}}$: Run the indistinguishability experiment $\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SURE}}(1^\lambda, 0)$.

$\mathbf{H}^0_{\text{proof}}$: Run the indistinguishability experiment as before, but replacing $\mathsf{Encode}(1^\lambda, C^0, x^0)$ with $\mathsf{Encode}^0_{\text{proof}}(1^\lambda, C^0, x^0, C^1, x^1)$ (see Figure 4), and replacing all executions of $\mathsf{GenUpd}(\mathsf{st}_i, \mathbf{u}^0_{i+1})$ with $\mathsf{GenUpd}_0(\mathsf{st}_i, \mathbf{u}^0_{i+1}, \mathbf{u}^1_{i+1})$ (see Figure 5).

$\mathbf{H}_0$: As above, except replacing $\mathsf{Encode}^0_{\text{proof}}(1^\lambda, C^0, x^0, C^1, x^1)$ with $\mathsf{Encode}^1_{\text{proof}}(1^\lambda, C^0, x^0, C^1, x^1)$. Here we still use GenUpd_0.

$\mathbf{H}_h$ for $1 \le h \le q$: As above, except using GenUpd_h.

$\mathbf{H}^1_{\text{real}}$: Run the indistinguishability experiment $\mathsf{Expt}_{\mathcal{A}}^{\mathsf{SURE}}(1^\lambda, 1)$. That is, replace $\mathsf{Encode}^1_{\text{proof}}(1^\lambda, C^0, x^0, C^1, x^1)$ with $\mathsf{Encode}(1^\lambda, C^1, x^1)$ and all executions of $\mathsf{GenUpd}_{\mathsf{q}}(\mathsf{st}_i, \mathbf{u}^0_{i+1}, \mathbf{u}^1_{i+1})$ with $\mathsf{GenUpd}(\mathsf{st}_i, \mathbf{u}^1_{i+1})$.

$\underline{\mathsf{Encode}^b_{\text{proof}}\left(1^\lambda, C^0, x^0, C^1, x^1\right):}$

On input security parameter λ, perform the following operations.

1. Execute the setup of FE, $\mathsf{FE.MSK} \leftarrow \mathsf{FE.Setup}(1^\lambda)$.
2. Compute a functional key $\mathsf{FE.SK} \leftarrow \mathsf{FE.KeyGen}(\mathsf{FE.MSK}, \mathsf{RRGarbler})$, where $\mathsf{RRGarbler}$ is as defined in Figure 3.
3. In the next step, generate a randomized encoding of input (C^b, x^b). That is, compute $\mathsf{RE.Enc}(1^\lambda, C^b, x^b)$ to obtain $\langle C^b[x^b]\rangle_{\mathsf{re}}$.
4. Generate a garbled circuit input encoding of $(C^0[x^0], C^1[x^1])$ by evaluating $\langle C^0[x^0], C^1[x^1]\rangle_{\mathsf{gc}} \leftarrow \mathsf{GrbInp}(C^0[x^0], C^1[x^1]; r_{\mathsf{gc}})$, where r_{gc} is the randomness used to garble the input.
5. Output as the randomized encoding the tuple $\langle C[x]\rangle_{\mathsf{ure}} = \Big(\mathsf{FE.SK},\ \langle C^b[x^b]\rangle_{\mathsf{re}},\ \langle C^0[x^0], C^1[x^1]\rangle_{\mathsf{gc}}\Big)$ and set the state to be $\mathsf{st} = (\mathsf{FE.MSK},\ r_{\mathsf{gc}})$.

Figure 4:

$\underline{\mathsf{GenUpd}_h\left(\mathsf{st}_i,\ \mathbf{u}^0_{i+1},\ \mathbf{u}^1_{i+1}\right)}$:

It takes as input the state $\mathsf{st}_i = (\mathsf{FE.MSK}, r_{\mathsf{gc},i})$ and update $\mathbf{u}_{i+1}$.

1. If $i+1 \leq h$, let $\mathsf{mode} = 1$. Otherwise, let $\mathsf{mode} = 0$.
2. Sample random coins $r_{\mathsf{re},i+1}$ and $r_{\mathsf{gc},i+1}$.
3. Generate the FE ciphertext $\mathsf{CT}_{i+1} \leftarrow \mathsf{FE.Enc}\left(\mathsf{FE.MSK}, \left(\mathbf{u}^0_{i+1}, \mathbf{u}^1_{i+1}, r_{\mathsf{gc},i}, r_{\mathsf{gc},i+1}, r_{\mathsf{re},i+1}, \mathsf{mode}\right)\right)$.
4. Set the new state $\mathsf{st}_{i+1} = (\mathsf{FE.MSK},\ r_{\mathsf{gc},i+1})$.
5. Output $\langle \mathbf{u}_{i+1} \rangle_{\mathsf{ure}} = \mathsf{CT}_{i+1}$.

Figure 5:

Indistinguishability of Hybrids. The differences between $\mathbf{H}^0_{\text{real}}$ and $\mathbf{H}^0_{\text{proof}}$ are exactly mirrored in the differences between hybrids $\mathbf{H}_q$ and $\mathbf{H}^1_{\text{real}}$, but with the bit $b \in \{0,1\}$ swapped in $\mathsf{Expt}^{\mathsf{SURE}}_{\mathcal{A}}(1^\lambda, b)$, Encode^b, and the setting of mode used by GenUpd. The indistinguishability of $\mathbf{H}_q$ and $\mathbf{H}^1_{\text{real}}$ follows essentially the same argument as we now present for $\mathbf{H}^0_{\text{real}}$ and $\mathbf{H}^0_{\text{proof}}$.

Moving from $\mathbf{H}^0_{\text{real}}$ to $\mathbf{H}^0_{\text{proof}}$ involves two changes which we take in turn. First, when generating the initial updatable randomized encoding, we switch from using $\mathsf{Encode}(1^\lambda, C^0, x^0)$ to using $\mathsf{Encode}^0(1^\lambda, C^0, x^0, C^1, x^1)$. The resulting change in the actual URE encoding is that initial garbled input is changed from $w = \langle C^0[x^0], \perp \rangle_{\mathsf{gc}}$ to $w' = \langle C^0[x^0], C^1[x^1] \rangle_{\mathsf{gc}}$. This causes cascading changes to the garbled inputs as each update is generated and applied, but everything else remains unchanged: the garbled $\mathsf{RelockRelease}_i$ circuits, the FE secret key, and all the RE encodings of the updated $C^0_i[x^0_i]$. We first observe that

$$\left(\langle C^0[x^0] \rangle_{\mathsf{re}}, w, \langle \mathsf{RelockRelease_1} \rangle_{\mathsf{gc}}, \ldots, \langle \mathsf{RelockRelease_q} \rangle_{\mathsf{gc}}\right)$$
$$\approx_c \left(\langle C^0[x^0] \rangle_{\mathsf{re}}, w', \langle \mathsf{RelockRelease_1} \rangle_{\mathsf{gc}}, \ldots, \langle \mathsf{RelockRelease_q} \rangle_{\mathsf{gc}}\right) \quad (1)$$

by a direct application of Lemma 2: the outputs of the cascaded garbled circuits are unchanged. Now a simple application of Lemma 1 establishes indistinguishability: Aux_0 and Aux_1 are the two tuples above respectively. The function f is $\mathsf{RRGarbler}$ and the messages M_i are those corresponding to the ciphertexts $\langle \mathbf{u}_i \rangle_{\mathsf{ure}} = \mathsf{FE.CT}_i$.

The second step in switching from $\mathbf{H}^0_{\text{real}}$ to $\mathbf{H}^0_{\text{proof}}$ is to switch GenUpd to GenUpd_0, ultimately changing each of the URE encoded updates from $\langle \mathbf{u}_i \rangle_{\mathsf{ure}} = \mathsf{FE.CT}_i$ to $\langle \mathbf{u}'_i \rangle_{\mathsf{ure}} = \mathsf{FE.CT}'_i$ containing both $\mathbf{u}^0_i$ and $\mathbf{u}^1_i$. While the FE key for $\mathsf{RRGarbler}$ is unchanged, the new ciphertexts result in output garbled circuits $\langle \mathsf{RelockRelease}'_i \rangle_{\mathsf{gc}}$, each of which now has both $\mathbf{u}^0_i$ and $\mathbf{u}^1_i$ hard-coded. This results in changes to the garbled input after each update is applied, but the all of the RE encodings $\langle C^0_i[x^0_i] \rangle_{\mathsf{re}}$ output by $\mathsf{RelockRelease}_i$ are exactly the same. Therefore, by Lemma 2, for each $j \in [q]$:

$$\left(\langle C^0[x^0] \rangle_{\mathsf{re}},\ w',\ \begin{array}{l} \langle \mathsf{RelockRelease_1} \rangle_{\mathsf{gc}}, \ldots, \langle \mathsf{RelockRelease}_j \rangle_{\mathsf{gc}}, \\ \langle \mathsf{RelockRelease}'_{j+1} \rangle_{\mathsf{gc}}, \ldots, \langle \mathsf{RelockRelease'_q} \rangle_{\mathsf{gc}} \end{array}\right)$$
$$\approx_c \left(\langle C^0[x^0] \rangle_{\mathsf{re}},\ w',\ \begin{array}{l} \langle \mathsf{RelockRelease_1} \rangle_{\mathsf{gc}}, \ldots, \langle \mathsf{RelockRelease}_{j-1} \rangle_{\mathsf{gc}}, \\ \langle \mathsf{RelockRelease}'_j \rangle_{\mathsf{gc}}, \ldots, \langle \mathsf{RelockRelease'_q} \rangle_{\mathsf{gc}} \end{array}\right)$$

For each j, we apply Lemma 1 to switch $\langle \mathbf{u}_j \rangle_{\mathsf{ure}}$ to $\langle \mathbf{u}'_j \rangle_{\mathsf{ure}}$, thereby establishing the indistinguishability of $\mathbf{H}^0_{\text{real}}$ and $\mathbf{H}^0_{\text{proof}}$.

The indistinguishability of $\mathbf{H}^0_{\text{proof}}$ and $\mathbf{H}_0$ is much simpler. Changing Encode^0 to Encode^1

switches the initial URE encoding from $\left(\mathsf{FE.SK}, \langle C^0[x^0]\rangle_{\mathsf{re}}, \langle C^0[x^0], C^1[x^1]\rangle_{\mathsf{gc}}\right)$ to $\left(\mathsf{FE.SK}, \langle C^1[x^1]\rangle_{\mathsf{re}}, \langle C^0[x^0], C^1[x^1]\rangle_{\mathsf{gc}}\right)$, and all URE encoded updates $\langle \mathbf{u}_i\rangle_{\mathsf{ure}}$ are unchanged. Notice that the randomness used to generate $\langle C^b[x^b]\rangle_{\mathsf{re}}$ is independent of everything in the adversary's view except for $\langle C^b[x^b]\rangle_{\mathsf{re}}$ itself. Given $\langle C^b[x^b]\rangle_{\mathsf{re}}$, along with C^0, C^1, x^0, and x^1, the view of the adversary can be exactly simulated. Therefore, the security of RE enables the switch from $\langle C^0[x^0]\rangle_{\mathsf{re}}$ to $\langle C^1[x^1]\rangle_{\mathsf{re}}$ that we require.

It remains now only to show for all $h \in [q-1]$ that $\mathbf{H}_h \approx_c \mathbf{H}_{h+1}$. In the latter hybrid, GenUpd_{h+1} is used, while GenUpd_h is used in the former. The only change in the view of the adversary is therefore the mode bit's setting in $\langle \mathbf{u}_{h+1}\rangle_{\mathsf{ure}} = \mathsf{FE.CT}_{h+1}$, which is set to 0 in the former and 1 in the latter. When evaluated using the FE functional key $\mathsf{FE.SK}$ for $\mathsf{RRGarbler}$, this results in the garbled $\mathsf{RelockRelease}_{h+1}$ outputting a randomized encoding $\langle C^1_{h+1}[x^1_{h+1}]\rangle_{\mathsf{re}}$ of $C^1_{h+1}[x^1_{h+1}]$ instead of $C^0_{h+1}[x^0_{h+1}]$; all other URE-encoded updates (FE ciphertexts) are unchanged. Recall in Defintion 14 we require that $C^0_{h+1}(x^0_{h+1}) = C^1_{h+1}(x^1_{h+1})$. This requirement, along with the security of RE, guarantees that

$$\begin{aligned}&\left(\langle C^1_0[x^1_0]\rangle_{\mathsf{re}}, \ldots, \langle C^1_h[x^1_h]\rangle_{\mathsf{re}}, \langle C^0_{h+1}[x^0_{h+1}]\rangle_{\mathsf{re}}, \langle C^0_{h+2}[x^0_{h+2}]\rangle_{\mathsf{re}}, \ldots, \langle C^0_q[x^0_q]\rangle_{\mathsf{re}}\right)\\ \approx_c &\left(\langle C^1_0[x^1_0]\rangle_{\mathsf{re}}, \ldots, \langle C^1_h[x^1_h]\rangle_{\mathsf{re}}, \langle C^1_{h+1}[x^1_{h+1}]\rangle_{\mathsf{re}}, \langle C^0_{h+2}[x^0_{h+2}]\rangle_{\mathsf{re}}, \ldots, \langle C^0_q[x^0_q]\rangle_{\mathsf{re}}\right)\end{aligned}$$

Applying Lemma 2 followed by Lemma 1 establishes indistinguishability of $\mathbf{H}_h$ and $\mathbf{H}_{h+1}$.

5 Updatable Garbled Circuits

In this section, we present definitions and a construction of updatable garbled circuits for the family of bit-wise updates. The main tool in our construction is a puncturable proxy re-encryption scheme, a notion that we define and construct in this section.

We start by giving a formal definition of updatable garbled circuits in Section 5.1. Next, in Section 5.2 we define and a construct a puncturable proxy re-encryption scheme based on puncturable (almost) key-homomorphic PRFs of [BV15b]. Finally, in Section 5.3, we give our construction of updatable garbled circuits for bit-wise updates from a puncturable proxy re-encryption scheme.

5.1 Definition of Updatable Garbled Circuits

Syntax. A scheme UGC $= (\mathsf{GrbCkt}, \mathsf{GrbInp}, \mathsf{GenUpd}, \mathsf{ApplyUpd}, \mathsf{EvalGC})$ for a $(\mathsf{Upd}, \mathcal{U})$-updatable class of circuits $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda\in\mathbb{N}}$ is defined below.

- **Circuit Garbling,** $\langle C\rangle_{\mathsf{gc}}, \mathsf{st} \leftarrow \mathsf{GrbCkt}(C \in \mathcal{C}_\lambda)$: On input circuit C, output the garbled circuit $\langle C\rangle_{\mathsf{gc}}$.
- **Generate Secure Update,** $\langle \mathbf{u}\rangle_{\mathsf{gc}}, \mathsf{st}' \leftarrow \mathsf{GenUpd}\,(\mathsf{st}, \mathbf{u} \in \mathcal{U}_\lambda)$: On input state st, update $\mathbf{u}$, output the secure update $\langle \mathbf{u}\rangle_{\mathsf{gc}}$.
- **Apply Secure Update,** $\langle C'\rangle_{\mathsf{gc}} \leftarrow \mathsf{ApplyUpd}\left(\langle C\rangle_{\mathsf{gc}}, \langle \mathbf{u}\rangle_{\mathsf{gc}}\right)$: On input the (old) garbled circuit $\langle C\rangle_{\mathsf{gc}}$, secure update $\langle \mathbf{u}\rangle_{\mathsf{gc}}$, output the updated garbled circuit $\langle C'\rangle_{\mathsf{gc}}$.
- **Input Garbling,** $\langle x\rangle_{\mathsf{gc}} \leftarrow \mathsf{GrbInp}(\mathsf{st}, x \in \{0,1\}^\lambda)$: On input state st, input $x \in \{0,1\}^\lambda$, output the garbled input $\langle x\rangle_{\mathsf{gc}}$.
- **Evaluation,** $\alpha \leftarrow \mathsf{EvalGC}(\langle C\rangle_{\mathsf{gc}}, \langle x\rangle_{\mathsf{gc}})$: On input garbled circuit $\langle C\rangle_{\mathsf{gc}}$, garbled input $\langle x\rangle_{\mathsf{gc}}$, output the result α.

Efficiency. We desire the same efficiency properties as in the definition of URE, except with the requirement on garbling (encoding) time modified appropriately to the setting of garbled circuits.

- *Garbling Time*: We require that the time to generate (and thus the size of) the garbled input $\langle x \rangle_{\mathsf{gc}}$ should be polynomial in $|x|$ and λ, and independent of the size of C and the number of updates. The time to generate the garbled circuit $\langle C \rangle_{\mathsf{gc}}$ should be polynomial in $|C|$ and λ.

- *Secure Update Generation Time*, *Secure Update Size*, *Runtime of Update*, and *State Size*: Same efficiency goals as URE.

While we would like to achieve all the efficiency goals simultaneously, in our construction of UGC presented in Section 5.3, the 'Runtime of Update' and the secret 'State Size' will both depend on $|C|$. Using the transformation described in Appendix C, it is possible to remove the dependence on $|C|$ for the state size.

Sequential Updating. The notion of sequential updating for updatable garbled circuits closely mirrors the URE notion. The key difference is that in the UGC setting, the encoded input is released only after all the encoded updates. Below, we define correctness and security of sequential updating.

Correctness of Sequential Updating. Let $C \in \mathcal{C}_\lambda, x \in \{0,1\}^\lambda$. Let $(\mathbf{u}_1, \ldots, \mathbf{u}_s) \in (\mathcal{U}_\lambda)^s$, where $s(\lambda)$ is a polynomial in λ. Consider the following two processes:

Secure updating process:

1. Let $\langle C \rangle_{\mathsf{gc}_0}, \mathsf{st}_0 \leftarrow \mathsf{GrbCkt}(C)$.
2. For $i \in [s]$, let $\langle \mathbf{u} \rangle_{\mathsf{gc}_i}, \mathsf{st}_i \leftarrow \mathsf{GenUpd}(\mathbf{u}_i, \mathsf{st}_{i-1})$
3. For $i \in [s]$, let $\langle C \rangle_{\mathsf{gc}_i} \leftarrow \mathsf{ApplyUpd}(\langle C \rangle_{\mathsf{gc}_{i-1}}, \langle \mathbf{u} \rangle_{\mathsf{gc}_i})$.
4. Let $\langle x \rangle_{\mathsf{gc}} \leftarrow \mathsf{GrbInp}(\mathsf{st}_s, x)$.

Insecure updating process:

1. Let $C_0 = C$.
2. For $i \in [s]$, let $C_i = \mathsf{Upd}(C_{i-1}, \mathbf{u}_i)$.

We have,

$$\mathsf{EvalGC}(\langle C \rangle_{\mathsf{gc}_s}, \langle x \rangle_{\mathsf{gc}}) = C_s(x)$$

IND-Security of Sequential Updating. As in the case of URE, we can consider two ways of defining the security notion. We only define the indistinguishability-based security notion below. The simulation-based definition can be appropriately defined. We denote the challenger by Ch and the PPT adversary by $\mathcal{A}$. The security is defined with respect to deterministic side-information functions ϕ_u, which capture what information is leaked by the encoded updates.

In our construction of updatable garbled circuits for bitwise updates $\mathbf{u} = (i, b)$, the side-information function $\phi_u(i, b) = i$ will leak the bit of the circuit representation being updated

$\mathsf{Expt}^{\mathrm{UGC}}_{\mathcal{A},\mathsf{Ind}}(1^\lambda, b \in \{0,1\})$:

- $\mathcal{A}$ sends circuits $C^0, C^1 \in \mathcal{C}_\lambda$, inputs $x^0, x^1 \in \{0,1\}^\lambda$ to Ch. Ch sends $\langle C^b \rangle_{\mathsf{ugc}}$, where $(\langle C^b \rangle_{\mathsf{ugc}}, \mathsf{st}_{\mathsf{ugc}}) \leftarrow \mathsf{GrbCkt}(1^\lambda, C^b)$, to $\mathcal{A}$.
- **Update Query Phase.** $\mathcal{A}$ then makes a series of update queries. For $i \in [q(\lambda)]$, it sends the ith pair of updates $(\mathbf{u}_i^0, \mathbf{u}_i^1)$ to Ch. If $\phi_u(\mathbf{u}_i^0) \neq \phi_u(\mathbf{u}_i^1)$, abort and return $\perp$. Ch sends $\langle \mathbf{u}_i^b \rangle_{\mathsf{ugc}}$ to $\mathcal{A}$ where $\langle \mathbf{u}_i^b \rangle_{\mathsf{ugc}}, \mathsf{st}_{\mathsf{ugc}}^i \leftarrow \mathsf{GenUpd}(\mathbf{u}_i^b, \mathsf{st}_{\mathsf{ugc}}^{i-1})$.
- Ch keeps local copies both circuits C^0 and C^1 up-to-date with respect to the updates queried. That is, letting $C_0^0 = C^0$ (respectively, C_0^1), Ch computes $C_i^0 \leftarrow \mathsf{Upd}(C_{i-1}^0, \mathbf{u}_i^0)$ (resp. C_i^1).
- If $C_q^0(x^0) \neq C_q^1(x^1)$, abort and return $\perp$. Otherwise, Ch sends $\mathsf{GrbInp}(\mathsf{st}_{\mathsf{ugc}}^q, \langle x^b \rangle_{\mathsf{ugc}})$ to the adversary.
- $\mathcal{A}$'s output is returned by $\mathsf{Expt}_{\mathcal{A},\mathsf{Ind}}^{\mathrm{UGC}}(1^\lambda, b)$.

We give the formal definition of the security notion below.

Definition 20. *A scheme* UGC *satisfies* **indistinguishability of sequential updating** *with leakage ϕ_u if for any PPT adversary $\mathcal{A}$ and some negligible function* negl

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

We say the scheme is **update-hiding** *if $\phi_u(u) = \perp$.*

We remark that our initial construction of UGC in Section 5.3 will not be update-hiding. In particular, to execute $\mathsf{ApplyUpd}$, the evaluator will have to know which bit of the circuit representation is being modified (though the old and new values of this bit will remain secret). In Appendix B, we describe a generic transformation from a non-update-hiding UGC scheme into one that achieves update hiding using a (non-interactive) oblivious RAM.

5.2 Puncturable Proxy Re-encryption Scheme

En route to constructing updatable garbled circuits, we introduce a tool called *puncturable symmetric proxy re-encryption scheme*. As in a standard proxy re-encryption scheme, our notion allows for generation of re-encryption keys. However, unlike a standard proxy re-encryption scheme, our notion allows for puncturing of re-encryption keys on ciphertexts such that the re-encryption mechanism fails on the punctured ciphertexts.

We begin by defining puncturable symmetric proxy re-encryption, along with identifying a number of additional properties our construction will satisfy that are useful in the construction of updatable garbled circuits. We then present a construction of such a re-encryption scheme from puncturable, almost key-homomorphic PRFs, which in turn can be based on hardness of approximating either GapSVP or SIVP to within sub-exponential factors.

5.2.1 Definition of Puncturable Symmetric Proxy Re-encryption

We give a formal definition of puncturable symmetric proxy re-encryption scheme below. This definition is tailored to our needs, and is not intended to be the most general definition of puncturable proxy re-encryption schemes.

Syntax A puncturable, symmetric proxy re-encryption scheme is a tuple of algorithms $\mathsf{ReEnc} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{ReKeyGen}, \mathsf{Enc}, \mathsf{ReEnc}, \mathsf{Dec})$ with the following syntax.

Setup, $\mathsf{PP} \leftarrow \mathsf{Setup}(1^\lambda)$: On input a security parameter λ in unary, output the public parameters PP of the ReEnc.

Key Generation, $\mathsf{SK} \leftarrow \mathsf{KeyGen}(\mathsf{PP})$: On input the public parameters PP, sample a ReEnc secret key SK.

Encryption, $\mathsf{CT} \leftarrow \mathsf{Enc}(\mathsf{SK}, \mathbf{m})$: On input a secret key SK and a message $\mathbf{m} \in \mathcal{M}$, output a ciphertext CT.

Decryption, $\mathbf{m} \leftarrow \mathsf{Dec}(\mathsf{SK}, \mathsf{CT})$: On input a secret key SK and a ciphertext CT, output a message $\mathbf{m} \in \mathcal{M}$.

Punctured Re-encryption Key Generation, $\mathsf{RK}^{\pi}_{1,2} \leftarrow \mathsf{ReKeyGen}(\mathsf{SK}_1, \mathsf{SK}_2, \pi)$: On input two secret keys SK_1 and SK_2 and a boolean-valued circuit π whose input is the space of ciphertexts, output a re-encryption key $\mathsf{RK}^{\pi}_{1,2}$. Informally, $\mathsf{RK}^{\pi}_{1,2}$ should only allow the re-encryption of ciphertexts CT for which $\pi(\mathsf{CT}) = 1$ to be re-encrypted from SK_1 to SK_2. We overload the notation $\mathsf{ReKeyGen}$, and will simply let $\mathsf{ReKeyGen}(\mathsf{SK}_1, \mathsf{SK}_2) := \mathsf{ReKeyGen}(\mathsf{SK}_1, \mathsf{SK}_2, \mathbf{1})$ where $\mathbf{1}$ is the constant 1 circuit.

Re-encryption, $\mathsf{CT}_2 \leftarrow \mathsf{ReEnc}(\mathsf{RK}_{1,2}\ , \mathsf{CT}_1)$: On input a re-encryption key $\mathsf{RK}_{1,2} \leftarrow \mathsf{ReKeyGen}(\mathsf{SK}_1, \mathsf{SK}_2)$ and a ciphertext $\mathsf{CT}_1 \leftarrow \mathsf{Enc}(\mathsf{SK}_1, \mathbf{m})$, output a ciphertext SK_2.

Informally, $\mathsf{RK}^{\pi}_{1,2}$ should only allow the re-encryption of ciphertexts CT for which $\pi(\mathsf{CT}) = 1$ to be re-encrypted from SK_1 to SK_2. We overload the notation $\mathsf{ReKeyGen}$, and will simply let $\mathsf{ReKeyGen}(\mathsf{SK}_1, \mathsf{SK}_2) := \mathsf{ReKeyGen}(\mathsf{SK}_1, \mathsf{SK}_2, \mathbf{1})$ where $\mathbf{1}$ is the constant 1 circuit.

For our application, we will consider only a restricted class of constraint circuits π, described immediately below.

Syntactic properties. Firstly, ciphertexts in ReEnc are of the form $\mathsf{CT} = (r, \mathsf{pl})$, where r is the randomness used in the encryption algorithm, and pl is the "payload." The randomness comes from a finite set $\mathcal{X}$ with description length polynomial in the security parameter.[4]

Secondly, re-encryption preserves the randomness of the ciphertext. That is, for any re-encryption key RK and ciphertext (r, pl), $\mathsf{ReEnc}(\mathsf{RK}, (r, \mathsf{pl})) = (r, \mathsf{pl}')$ for some pl'.

Lastly, we restrict our attention to the constraint circuits of the following form:

$$\pi_{r^*}(r, \mathsf{pl}) = \begin{cases} 0 & \text{if } r = r^* \\ 1 & \text{if } r \neq r^* \end{cases}$$

We abuse notation and identify r^* with the constraint $\pi(r^*)$, and additionally primitives with such constraints "punctured at r^*."

Correctness. The re-encryption scheme ReEnc satisfies a modified correctness property that reflects the informal goal that re-encryption keys punctured at r^* should enable re-encryption only of ciphertexts with randomness $r \neq r^*$.

Definition 21 (Ciphertext Punctured Re-encryption: Correctness)**.** *We will say the ciphertext constrained re-encryption scheme* ReEnc *is correct if, under public parameters* $\mathsf{PP} \leftarrow \mathsf{Setup}(1^\lambda)$*, secret key* $\mathsf{SK} \leftarrow \mathsf{KeyGen}(1^\lambda)$*, and for all* $\mathbf{m} \in \mathcal{M}$*, two conditions hold:*

- $\mathsf{Dec}(\mathsf{SK}, \mathsf{Enc}(\mathsf{SK}, \mathbf{m})) = \mathbf{m}$
- *For all* $T = \mathrm{poly}(\lambda)$*, and for any sequence of secret keys* $\mathsf{SK}_1, \ldots, \mathsf{SK}_T \leftarrow \mathsf{KeyGen}(1^\lambda)$*, and re-encryption keys* $\mathsf{RK}^{r_i}_{i,i+1} \leftarrow \mathsf{ReKeyGen}(\mathsf{SK}_i, \mathsf{SK}_{i+1}, r_i)$ *for all* $i \in [1, T-1]$*, for all messages* $\mathbf{m} \in \mathcal{M}$*, and all sequences of ciphertexts* $\mathsf{CT}_1 \leftarrow \mathsf{Enc}(\mathsf{SK}_1, \mathbf{m})$*,* $\mathsf{CT}_2 \leftarrow \mathsf{ReEnc}(\mathsf{RK}^{r_2}_{1,2}, \mathsf{CT}_1)$*, …,* $\mathsf{CT}_T \leftarrow \mathsf{ReEnc}(\mathsf{RK}^{r_{T-1}}_{T-1,T}, \mathsf{CT}_{T-1})$ *it holds that either:*
  1. $\mathsf{Dec}(\mathsf{SK}_T, \mathsf{CT}_T) = \mathbf{m}$*, or*
  2. $\exists i \in [1, T-1]$ *such that* $r = r_i$*, where* r *is the randomness of the ciphertext* CT_1 *(and thus of all ciphertexts* CT_j*).*

[4]Instantiating the PRF as we do, $\mathcal{X}$ will be the set of binary strings of appropriate length.

Security. The semantic security definition of [BLMR13] does not suffice for our needs. We require a different, incomparable security definition that captures the power of the constrained re-encryption keys. In formalizing our security requirement, we diverge significantly from [BLMR13]. A security definition that both suffices for the application to updatable garbled circuits and generalizes the [BLMR13] definition would unnecessarily complicate our discussion and is outside the scope of this work. Below, we present a security definition that is tailored to our requirements.

Informally, we wish to guarantee security when the adversary gets constrained re-encryption keys *and a terminal secret key*, if the constraint prevents "honest re-encryption" of the challenge ciphertext.[5] In a very simplified form, this setting can be seen in the following game between an adversary $\mathcal{A}$ and a challenger Ch:

- $\mathcal{A}$ chooses messages $\mathbf{m}^0$ and $\mathbf{m}^1$.
- Ch chooses a random bit $b \leftarrow \{0,1\}$, two random secret keys $\mathsf{SK}_1, \mathsf{SK}_2 \leftarrow \mathsf{KeyGen}(1^\lambda)$, and sends $c^* = (r^*, \mathsf{pl}^*) \leftarrow \mathsf{Enc}(\mathsf{SK}_1, \mathbf{m}^b)$.
- Ch generates $\mathsf{RK}_{1,2}^{r^*}$, and sends both $\mathsf{RK}_{1,2}^{r^*}$ and SK_2 to $\mathcal{A}$.
- $\mathcal{A}$ attempts to guess b with probability significantly better than $1/2$.

Our actual security definition is somewhat more complex. The adversary will receive encryptions of many messages, a chain of punctured re-encryption keys, and a single secret key. We also introduce the additional complexity that the messages may originally be encrypted with any of the secret keys $\mathsf{SK}_0, \dots, \mathsf{SK}_q$. Lastly, we must impose a non-triviality condition – which we term "validity" – described in the definition and further discussed below.

$\underline{\mathsf{Expt}_{\mathcal{A}}^{\mathsf{ReEnc}}(1^\lambda, b \in \{0,1\})}$:

- The challenger samples and sends $\mathsf{PP} \leftarrow \mathsf{Setup}(1^\lambda)$ to $\mathcal{A}$.
- The adversary selects two sequences of messages $(\mathbf{m}_1^0, \dots, \mathbf{m}_\ell^0)$ and $(\mathbf{m}_1^1, \dots, \mathbf{m}_\ell^1)$ for some $\ell = \mathrm{poly}(\lambda)$, along with a sequence of q "punctured re-encryption key requests:" $(p_1, \dots, p_q) \in [\ell]$, where each p_j is distinct. Additionally, the adversary specifies a tuple of key-indices $(k_1, \dots, k_\ell) \in [q]$.
- The challenger checks the validity condition. If it is violated, the experiment aborts.
  - For all $i \in [\ell]$, if $\forall j \ : \ \big(p_j \neq i\big) \vee \big(j < k_i\big)$, then $\mathbf{m}_i^0 = \mathbf{m}_i^1$.
- The challenger samples secret keys $\mathsf{SK}_i \leftarrow \mathsf{KeyGen}(1^\lambda)$ for $i \in [q+1]$, and sends the following to the adversary:
  - Ciphertexts $\mathsf{CT}_i \leftarrow \mathsf{Enc}(\mathsf{SK}_{k_i}, \mathbf{m}_i^b)$, for all $i \in [\ell]$. Let r_i be the randomness used in CT_i.
  - Punctured re-encryption keys $\mathsf{RK}_{j,j+1}^{r_{p_j}} \leftarrow \mathsf{ReKeyGen}(\mathsf{SK}_j, \mathsf{SK}_{j+1}, r_{p_j})$, for each $j \in [q]$, punctured to not work on CT_{p_j}.
  - The final secret key SK_{q+1}.
- The adversary outputs b'. The output of the experiment is 1 if $b' = b$, and 0 if $b' \neq b$.

Definition 22. *A scheme* ReEnc *is* secure *if for any PPT adversary* $\mathcal{A}$ *and some negligible function* negl

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

Notice that the definition would be trivially unsatisfiable without the validity check. Unless a re-encryption key after k_i is punctured at the ciphertext, then given the re-encryption keys along with SK_q, the adversary can simply recover $\mathbf{m}_i^b$. Therefore, we require the additional condition that for any such i, $\mathbf{m}_i^0 = \mathbf{m}_i^1$.

[5] Corresponding to the last condition in the correctness properties above.

Fresh-stale indistinguishability Our scheme will enjoy one more security property: that for secret keys SK_1 and SK_2 with re-encryption key $\mathsf{RK}_{1,2}$, "fresh" ciphertexts produced with SK_2 are indistinguishable from ciphertexts (of the same message) encrypted under SK_1 and then re-encrypted using $\mathsf{RK}_{1,2}$, even *given* SK_1 *and* SK_2.

Definition 23 (Fresh-stale indistinguishability)**.** ReEnc *is said to satisfy* fresh-stale indistinguishability *if, for all messages* $\mathbf{m}$ *and randomness* r^**:*

$$\Big(\mathsf{Enc}(\mathsf{SK}_2, \mathbf{m}),\ \mathsf{SK}_1,\ \mathsf{SK}_2,\ \mathsf{PP},\ \mathbf{m},\ r^*\Big) \approx_c \Big(\mathsf{ReEnc}(\mathsf{RK}^{r^*}_{1,2}, \mathsf{Enc}(\mathsf{SK}_1, \mathbf{m})),\ \mathsf{SK}_1,\ \mathsf{SK}_2,\ \mathsf{PP},\ \mathbf{m},\ r^*\Big)$$

where the probabilities are taken over $\mathsf{PP} \leftarrow \mathsf{Setup}(1^\lambda)$*,* $\mathsf{SK}_1, \mathsf{SK}_2 \leftarrow \mathsf{KeyGen}$*,* $\mathsf{RK}^{r^*}_{1,2} \leftarrow \mathsf{ReKeyGen}(\mathsf{SK}_1, \mathsf{SK}_2, r^*)$*, along with* Enc *and* ReEnc*.*

Note that by a hybrid argument, fresh-stale indistinguishability as we defined for SK_1 and SK_2 can be extended for any polynomial-length chain of secret keys. That is, a fresh encryption under SK_q is indistinguishable from q-many re-encryptions of a ciphertext through the keys $\mathsf{SK}_1,\ldots,\mathsf{SK}_{q+1}$, even when given all the keys $\mathsf{SK}_1, \ldots, \mathsf{SK}_{q+1}$.

Looking ahead, fresh-stale indistinguishability will be useful in proving security (Definition 22). In Section 5.2.3, we construct a puncturable proxy re-encryption scheme as described in this section from almost key-homomorphic puncturable PRFs. If we instead instantiated our construction with (perfectly) key homomorphic PRFs, then our scheme would immediately satisfy fresh-stale indistinguishability (in fact, the distributions would be equivalent).

5.2.2 Building Block: Puncturable, Almost Key-Homomorphic PRFs

We use puncturable almost key-homomorphic PRFs [BLMR13, BV15b] to build updatable garbled circuits. In [BLMR13], the authors construct a symmetric proxy re-encryption scheme from any almost key-homomorphic PRF. We similarly construct a *puncturable*, symmetric proxy re-encryption scheme given a *puncturable* almost key-homomorphic PRF, which can be based on worst-case lattice assumptions as shown in the recent work of [BV15b]. We now present a definition of puncturable, almost key-homomorphic PRFs adapted from [BLMR13] and recall the main theorem of [BV15b]. We restrict our attention to PRFs with the particular co-domain $\mathbb{Z}_p$ for some integer and p.

Definition 24 (Puncturable, γ-Almost Key Homomorphic PRF)**.** *Let* $\mathsf{PRF} = (\mathsf{PRF.Setup}, \mathsf{PRF.KeyGen}, \mathsf{PRF.Punct}, F)$ *be point-puncturable PRF family,*[6] *with keyspace* $\mathcal{K}_\lambda$*, domain* $\mathcal{X}_\lambda$*, and co-domain* $\mathbb{Z}_p$*, for* $\lambda \in \mathbb{N}$*, and some integer* p*. Suppose that* $(\mathcal{K}_\lambda, +)$ *and* $(\mathcal{Y}, +)$ *are groups. The puncturable PRF family is* γ-almost key homomorphic *if for all points* x_1^**, and* x_2^**, and all keys* K_1 *and* K_2*, and all inputs* $x \neq X_1^*, x^*, 2$*, there exists an* $e \in [0, \gamma]$ *such that*

$$F(\mathsf{PRF.Punct}(K_1, x_1), x) + F(\mathsf{PRF.Punct}(K_2, x_2^*), x) = F(K_1 + K_2, x) \pm e \pmod{p}.$$

The security requirements of punctured PRFs vary from construction to construction (e.g. the number of punctured keys, the selectivity / adaptivity of punctured points). In this work we consider a weak security notion: selective security with respect to only a single punctured key. Such PRFs were constructed in [BV15b].

Theorem 7 ([BV15b], Theorem 5.1)**.** *For* $c > 0$*, a single-key, selectively secure point-puncturable, 2-almost key homomorphic PRF with domain* $\{0,1\}^n$ *and co-domain* $\mathbb{Z}_p$ *can be constructed based on the hardness of approximating either* GapSVP *or* SIVP *to within a factor of* $2^{\widetilde{O}(n^{1/c})}$*, where* $n = (\lambda \log \lambda)^c$*,* $p = 2^{\widetilde{O}(n^{1/c})}$*.*

[6]A puncturable PRF family is a constrained PRF family (see [BW13, BV15b]) for the family of constraints which are equal to 1 except at a single "punctured" point.

Proof. We observe that the family of circuits that check whether an input $r \in \{0,1\}^\lambda$ is equal to a specific r^* can be uniformly generated, and each such circuit has depth $O(\log \lambda)$ and a description of size λ (the description being r^* itself).

Applying the choice of parameters in Section 5.2 of [BV15b], along with the observations in Section 5.5 of that work yields the corollary. □

Finally, we remark that this construction has a property that will be important for us in the proof of security for ReEnc. Namely, for all $\mathsf{PP} \leftarrow \mathsf{PRF.Setup}(1^\lambda)$, the following two distributions over the keyspace $\mathcal{K}_\lambda$ are identical:

$$\mathsf{PRF.KeyGen}(\mathsf{PP}) \equiv \mathsf{Uniform}(\mathcal{K}_\lambda) \tag{2}$$

5.2.3 Construction of Puncturable Symmetric Proxy Re-encryption

We now present a puncturable re-encryption scheme that satisfies all the properties discussed in Section 5.2.1. Our construction is closely modeled on [BLMR13].[7] The main ingredient in our construction is a puncturable almost key-homomorphic PRF scheme that additionally satisfies the property stated in equation 2. Using the PRF scheme from the previous section, we state our main theorem for this section.

Theorem 8. *Assuming the hardness of approximating either* GapSVP *or* SIVP *to within sub-exponential factors, there exists a symmetric puncturable re-encryption scheme.*

The rest of this subsection is dedicated to the proof of this theorem.

Construction. Let PRF be the family of puncturable, 2-almost key homomorphic PRFs from Corollary 7. The domain and range of PRF are $\{0,1\}^n$ and $\mathbb{Z}_p$ respectively, where $n = \text{poly}(\lambda)$. Let $B = \lambda^{\log \lambda}$ and let u be an integer such that $\lfloor p/u \rfloor > 6B$ (recall that $p = 2^{\widetilde{O}(n^{1/c})}$).

Construction We define $\mathsf{ReEnc} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{ReKeyGen}, \mathsf{Enc}, \mathsf{ReEnc}, \mathsf{Dec})$ to be an encryption scheme with message space $\mathbb{Z}_u$ as follows:

- $\mathsf{Setup}(1^\lambda)$: Sample and output the public parameters of the PRF $\mathsf{PP} \leftarrow \mathsf{PRF.Setup}(1^\lambda)$.
- $\mathsf{KeyGen}(1^\lambda)$: Output a secret key $\mathsf{SK} \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$.
- $\mathsf{Enc}(\mathsf{SK}, \mathbf{m})$: Sample $r \leftarrow \{0,1\}^n$ and noise $\eta \leftarrow [-B, B]$, and output (r, pl) where $\mathsf{pl} = m \cdot \lfloor p/u \rfloor + F(\mathsf{SK}, r) + \eta$.
- $\mathsf{Dec}(\mathsf{SK}, (r, \mathsf{pl}))$: Output $\lfloor \mathsf{pl} - F(\mathsf{SK}, r) \pmod p \rceil_u$, where $\lfloor \cdot \rceil_u$ denotes rounding to the nearest multiple of $\lfloor p/u \rfloor$.
- $\mathsf{ReKeyGen}(\mathsf{SK}_1, \mathsf{SK}_2, r^*)$: Output $\mathsf{RK}_{1,2}^{r^*} \leftarrow \mathsf{PRF.Punct}(\mathsf{SK}_2 - \mathsf{SK}_1, r^*)$.
- $\mathsf{ReEnc}(\mathsf{RK}, (r, \mathsf{pl}))$: Output $(r, \mathsf{pl} + F(\mathsf{RK}, r))$.

Correctness Correctness follows in a straightforward manner from the γ-almost key-homomorphic guarantee of PRF and the choice of u. This is because the magnitude of the total accumulated error (from $T = \text{poly}(\lambda)$ re-encryptions and the encryption noise η) cannot exceed $(2T + B) < 3B$ and thus the error does not affect decryption correctness.

[7]As in that work, the scheme would be simplified by using key homomorphic PRFs, rather than *almost* key homomorphic PRFs. Because only the latter are known to exist from lattice assumptions, we present only that construction.

Fresh-stale indistinguishability We wish to show that for any $\mathbf{m}$, SK_1, SK_2, and r^* it is infeasible to determine whether a given ciphertext $\mathsf{CT} = (r, \mathsf{pl})$ was generated according to $\mathsf{CT} = \mathsf{CT}_2 \leftarrow \mathsf{Enc}(\mathsf{SK}_2, \mathbf{m})$, or according to $\mathsf{CT} = \mathsf{CT}_1 \leftarrow \mathsf{ReEnc}(\mathsf{ReKeyGen}(\mathsf{SK}_1, \mathsf{SK}_2, r^*), \mathsf{Enc}(\mathsf{SK}_1, \mathbf{m}))$. Note that in our scheme, both $\mathsf{ReKeyGen}$ and ReEnc are deterministic algorithms. Intuitively, the error from re-encrypting the ciphertext instead of computing it fresh is drowned out by the noise η added to the ciphertext during encryption.

Let $c, B, \Delta \in \mathbb{Z}_p$ such that $B < p/4$ and $c < p/4$. Let D_2 be the uniform distribution over $[c-B, +B]$, and let D_1 be uniform over $[c-B+\Delta, c+B+\Delta]$. The statistical distance of these distributions is $\mathsf{SD}(D_1, D_2) = \Delta/(2B+1)$.

Fix any choice of randomness r for Enc, and any choice of SK_1, SK_2, r^* and m. Let $c = m \cdot \lfloor p/u \rfloor + F(\mathsf{SK}_2, r)$ and $\Delta = |F(\mathsf{SK}_2, r) - (F(\mathsf{SK}_1, r) + F(\mathsf{PRF.Punct}(\mathsf{SK}_2 - \mathsf{SK}_1, r^*), r))| \leq 2\gamma$. Consider the distribution (induced by the choice of noise $\eta \in [-B, B]$) over ciphertexts $\mathsf{CT}_1 = (r, \mathsf{pl}_1)$ and $\mathsf{CT}_2 = (r, \mathsf{pl}_2)$ defined as above. CT_1 is distributed according to D_1 and CT_2 is distributed according to D_2. Therefore, the distributions over CT_1 and CT_2 induced by the choice of noise η by Enc have statistical distance bounded by $2\gamma/(2B+1)$. Because B is chosen to be superpolynomial, this statistical distance is negligible.

Security Suppose for contradiction that there existed an adversary $\mathcal{A}$ for which $\mathsf{Expt}_{\mathcal{A}}^{\mathsf{ReEnc}}(1^\lambda, 0)$ and $\mathsf{Expt}_{\mathcal{A}}^{\mathsf{ReEnc}}(1^\lambda, 1)$ were distinguishable. First, we observe that fresh-stale indistinguishability implies that, without loss of generality we may assume that for all $i \in [\ell]$, $k_i = 0$ (i.e. all the messages $\mathbf{m}_i^b$ are originally encrypted under SK_1 instead of SK_{k_i}). This follows directly from the observation that fresh-stale indistinguishability extends to chains of secret keys of polynomial length (see note after Definition 23), along with the requirement that if $p_j = i$, then $j > i$. If some $k_i > 0$, then the view of that adversary can be indistinguishably simulated by using $k_i' = 1$, getting the encryption of $\mathbf{m}_i^b$, then using the re-encryption keys to compute a ciphertext with respect to key SK_{k_i}. Because the re-encryption keys $\mathsf{RK}_{1,2}, \ldots, \mathsf{RK}_{i-1,i}$ are not punctured at the ciphertext CT_i (by the restriction on p_j previously mentioned), this simulation will succeed.

Second, by a simple hybrid argument, we may also assume without loss of generality that the sequences of messages $(\mathbf{m}_1^0, \ldots, \mathbf{m}_\ell^0)$ and $(\mathbf{m}_1^1, \ldots, \mathbf{m}_\ell^1)$ differ at exactly 1 index i^*. For all $i \neq i^*$, let $\mathbf{m}_i = \mathbf{m}_i^0 = \mathbf{m}_i^1$. We may also assume that there exists $j^* \in [q]$ such that $p_{j^*} = i^*$; otherwise $\mathcal{A}$'s challenge is invalid and the challenger aborts.

We use $\mathcal{A}$ to violate the single-key, selective security of the punctured PRF, yielding a contradiction. Given the public parameters PP of PRF, sample a uniform PRF input r^*, and receive in turn the punctured PRF challenge: a punctured key K^{r^*} and y^*, which is either $F(K, r^*)$ or a uniformly random value in the co-domain $\mathbb{Z}_p$.

Pass along the public parameters to $\mathcal{A}$ and receive in return $(\mathbf{m}_1, \ldots, \mathbf{m}_{i^*-1}, \mathbf{m}_{i^*}^0, \mathbf{m}_{i^*}^1, \mathbf{m}_{i^*+1}, \ldots, \mathbf{m}_\ell)$, and $(p_1, \ldots, p_q)$ with $p_{j^*} = i^*$.

For $j \neq j^*$, sample PRF keys $\mathsf{RK}_{j,j+1} \leftarrow \mathsf{PRF.KeyGen}(\mathsf{PP})$ (which we identify with re-encryption keys), and a final PRF key $\mathsf{SK}_{q+1} \leftarrow \mathsf{PRF.KeyGen}(\mathsf{PP})$ (interpreted as the terminal secret key in the ReEnc security game). Lastly, set $\mathsf{RK}_{j^*,j^*+1}^{r^*} = K^{r^*}$, the challenge punctured PRF key.

For each $i \neq i^*$, sample encryption randomness r_i uniformly and compute the ciphertext $\mathsf{CT}_i^{q+1} = \mathsf{Enc}(\mathsf{SK}_{q+1}, \mathbf{m}_i; r_i)$. Then, compute

$$\mathsf{CT}_i = \mathsf{CT}_i^{q+1} - F(\mathsf{RK}_{j^*,j^*+1}^{r^*}, r_i) - \sum_{j \neq j^*} F(\mathsf{RK}_{j,j+1}, r_i) \tag{3}$$

For i^*, pick $b \leftarrow \{0,1\}$ uniformly, and compute the ciphertext $\mathsf{CT}_i^{q+1} = \mathsf{Enc}(\mathsf{SK}_{q+1}, \mathbf{m}_i^b; r_i)$. Then compute

$$\mathsf{CT}_{i^*} = \mathsf{CT}_{i^*}^{q+1} - y^* - \sum_{j \neq j^* 1} F(\mathsf{RK}_{j,j+1}, r_{i^*})$$

Finally, compute the punctured re-encryption keys $\mathsf{RK}_{j,j+1}^{r_{p_j}} \leftarrow \mathsf{PRF.Punct}(\mathsf{RK}_{j,j+1}, r_{p_j})$.

As in the security definition for punctured proxy re-encryption, return to $\mathcal{A}$ the CT_i for all i, $\mathsf{RK}_{j,j+1}^{r_{p_j}}$ for all j, and SK_{q+1}. By Equation 2, the re-encryption keys $\mathsf{RK}_{j,j+1}^{r_{p_j}}$ and the secret key SK_{q+1} are distributed exactly as in the actual ReEnc security game. On the other hand, the ciphertexts CT_i are not distributed like honest ciphertexts, even for $i \neq i^*$. This is because the PRF is only *almost* key-homomorphic – there's no guarantee that $F(\sum \mathsf{RK}_{j,j+1}, r_i)$ is indistinguishable from $\sum F(\mathsf{RK}_{j,j+1}, r_i)$ when also given the (punctured) re-encryption keys.

We again use fresh-stale indistinguishability. If instead of generating the CT_i^{q+1} as we did by encrypting directly using SK_{q+1}, the CT_i^{q+1} had been generated as re-encryptions of ciphertexts $\mathsf{CT}_i \leftarrow \mathsf{Enc}(\mathsf{SK}_1, \mathbf{m}_i)$ using the punctured re-encryption keys (i.e. if the CT_i^{q+1} were "stale"), then the transformation we compute in (3) would exactly recover the fresh ciphertexts CT_i. On the other hand, the reduction used in the proof uses "fresh" ciphertexts CT_i^{q+1}. Therefore an adversary $\mathcal{A}$ who could distinguish its view in the proof, where the CT_i are generated as in (3) from fresh encryptions $\mathsf{CT}_i^{q+1} \leftarrow \mathsf{Enc}(\mathsf{SK}_{q+1}, \mathbf{m}_i)$ could be used to violate fresh-stale indistinguishability. Given a fresh-or-stale ciphertext CT_i^{q+1} and all the secret keys SK_j, it is easy to generate the re-encryption keys in the view of the adversary.

Therefore, we are able to conclude that the view of the adversary in this proof of ReEnc security is indistinguishable from the real-world view, and the adversary (by assumption) must succeed with non-negliglbe probability. Now, if punctured PRF challenge value $y^* = F(K, r^*)$, then the view of the adversary is indistinguishable from its view in $\mathsf{Expt}_{\mathcal{A}}^{\mathsf{ReEnc}}(1^\lambda, b)$ with random b. On the other hand, if y^* is random, then CT_{i^*} is distributed uniformly, and hides the value of the bit b. Therefore, if $\left|\Pr[1 \leftarrow \mathsf{Expt}_{\mathcal{A}}^{\mathsf{ReEnc}}(1^\lambda, 0)] - \Pr[1 \leftarrow \mathsf{Expt}_{\mathcal{A}}^{\mathsf{ReEnc}}(1^\lambda, 1)]\right|$ is non-negliglbe, we are able to distinguish the case when $y^* = F(K, r^*)$ or is random, violating the security of PRF and completing the proof.

5.3 Construction of UGC

We now present our construction of UGC for general circuits and the family of bit-wise updates. We proceed in two steps:

1. First, we present a construction of UGC for general circuits and bit-wise updates that does not achieve update hiding.
2. In Appendix B, we present a generic transformation from a non-update-hiding UGC scheme into one that achieves update hiding. Applying this transformation on our UGC scheme constructed in the first step, we obtain an update-hiding UGC scheme for general circuits and bit-wise updates.

We prove the following theorem in this Section:

Theorem 9. *Suppose* ReEnc *is a puncturable symmetric proxy re-encryption scheme and* GC *is an efficient* $\mathsf{PrivIND}_\phi$*-secure projective garbling scheme for all circuits. Then there exists an IND-secure, sequentially updatable garbled circuits scheme* UGC *for the class* $\mathcal{C}$ *of updatable circuits with bitwise updates* $(\mathsf{Upd}_{\mathsf{bit}}, \mathcal{U}_{\mathsf{bit}})$*, which is not update-hiding.*

Applying the transformation in Appendix B, we obtain our main result of UGC:

Corollary 1. *Under the same assumptions as Theorem 9, there is an update-hiding, IND-secure, sequentially updatable garbled circuits scheme* UGC *for the same class of circuits and updates.*

The rest of this Section is devoted to the proof of Theorem 9.

5.3.1 Proof of Theorem 9

Let ReEnc be a puncturable symmetric proxy re-encryption scheme with the properties outlined in Section 5.2.1, and let GC be an efficient, $\mathsf{PrivIND}_\phi$-secure projective garbling scheme for all circuits (Definition 3). We construct a non-update-hiding updatable garbling scheme UGC = $(\mathsf{GrbCkt}, \mathsf{GrbInp}, \mathsf{GenUpd}, \mathsf{ApplyUpd}, \mathsf{EvalGC})$ as follows.

$\underline{\mathsf{GrbCkt}(1^\lambda, C)}$: On input a circuit C, do the following.

1. Let U be the universal circuit of size, input- and output-length of C, and let $\mathsf{binary}(C)$ be the corresponding representation of C. Let $s = |\mathsf{binary}(C)|$ be the length of the description of C.
2. Generate the garbled circuit and state $(\langle U \rangle_{\mathsf{gc}}, \mathsf{st}_{\mathsf{gc}}) \leftarrow \mathsf{GC.GrbCkt}(U)$.
3. Using $\mathsf{st}_{\mathsf{gc}}$, generate the garbled input wire labels corresponding to the circuit description $\mathsf{binary}(C)$: $\langle \mathsf{binary}(C) \rangle_{\mathsf{gc}} \leftarrow \mathsf{GC.GrbInp}(\mathsf{st}_{\mathsf{gc}}, \mathsf{binary}(C))$.[8] Write $\langle \mathsf{binary}(C) \rangle_{\mathsf{gc}}$ as $\langle \mathsf{binary}(C)_1 \rangle_{\mathsf{gc}}, \ldots, \langle \mathsf{binary}(C)_s \rangle_{\mathsf{gc}}$.
4. Sample randomness $\mathbf{r}_{\mathsf{reEnc}} = (r_1, \ldots, r_s)$ for ReEnc encryption. Sample public parameters $\mathsf{PP} \leftarrow \mathsf{ReEnc.Setup}(1^\lambda)$ and secret key $\mathsf{SK} \leftarrow \mathsf{ReEnc.KeyGen}(\mathsf{PP})$ for the re-encryption scheme.
5. For each $i \in [s]$, encrypt $\mathsf{CT}_i = \mathsf{Enc}(\mathsf{SK}, \langle \mathsf{binary}(C)_i \rangle_{\mathsf{gc}};\ r_i)$, using r_i as the encryption randomness.
6. Output $\langle C \rangle_{\mathsf{ugc}} := (\langle U \rangle_{\mathsf{gc}}, \mathsf{PP}, \mathsf{CT}_1, \ldots, \mathsf{CT}_s)$ and $\mathsf{st}_{\mathsf{ugc}} = (\mathsf{st}_{\mathsf{gc}}, \mathsf{PP}, \mathsf{SK}, \mathbf{r}_{\mathsf{reEnc}})$.

$\underline{\mathsf{GenUpd}(\mathsf{st}_{\mathsf{ugc}}, \mathbf{u})}$: On input a state $\mathsf{st} = (\mathsf{st}_{\mathsf{gc}}, \mathsf{SK}, \mathbf{r}_{\mathsf{reEnc}})$ and update $\mathbf{u} = (i, b)$, do the following.

1. Sample a fresh secret key $\mathsf{SK}' \leftarrow \mathsf{KeyGen}(1^\lambda)$. Generate the re-encryption key $\mathsf{RK} \leftarrow \mathsf{ReKeyGen}(\mathsf{SK}, \mathsf{SK}', r_i)$, punctured at r_i, the ith element of $\mathbf{r}_{\mathsf{reEnc}}$.
2. Sample new randomness r'_i for $\mathsf{ReEnc.Enc}$. Let $\mathbf{r}'_{\mathsf{reEnc}}$ be $\mathbf{r}_{\mathsf{reEnc}}$ with r_i replaced by r'_i.
3. Compute the new garbled wire input label $L_{i,b} = \mathsf{GC.GrbInputWire}(\mathsf{st}, i)_b$, and encrypt the new input wire label $\mathsf{CT}'_i = \mathsf{Enc}(\mathsf{SK}, L_{i,b}; r'_i)$.
4. Output the garbled update $\langle \mathbf{u} \rangle_{\mathsf{ugc}} := (i, \mathsf{CT}'_i, \mathsf{RK}^{r_i})$ and the updated state $\mathsf{st}'_{\mathsf{ugc}} = (\mathsf{st}_{\mathsf{gc}}, \mathsf{SK}', \mathbf{r}'_{\mathsf{reEnc}})$.

$\underline{\mathsf{ApplyUpd}(\langle C \rangle_{\mathsf{ugc}}, \langle \mathbf{u} \rangle_{\mathsf{ugc}})}$: On input a UGC-garbled circuit $\langle C \rangle_{\mathsf{ugc}} = (\langle U \rangle_{\mathsf{gc}}, \mathsf{CT}_1, \ldots, \mathsf{CT}_s)$ and a garbled update $\langle \mathbf{u} \rangle_{\mathsf{ugc}} = (i, \mathsf{CT}'_i, \mathsf{RK})$:

1. For each $j \neq i$, let $\mathsf{CT}'_j = \mathsf{ReEnc}(\mathsf{RK}, \mathsf{CT}_j)$.
2. Output $\langle C \rangle'_{\mathsf{ugc}} := (\langle U \rangle_{\mathsf{gc}}, \mathsf{CT}'_1, \ldots, \mathsf{CT}'_s)$

$\underline{\mathsf{GrbInp}(\mathsf{st}_{\mathsf{ugc}}, x)}$: On input UGC state $\mathsf{st}_{\mathsf{ugc}} = (\mathsf{st}_{\mathsf{gc}}, \mathsf{SK}, \mathbf{r}_{\mathsf{reEnc}})$ and an input $x \in \{0,1\}^\lambda$:

1. Using $\mathsf{st}_{\mathsf{gc}}$, generate the garbled input $\langle x \rangle_{\mathsf{gc}} \leftarrow \mathsf{GC.GrbInp}(\mathsf{st}_{\mathsf{gc}}, x)$.
2. Return $\langle x \rangle_{\mathsf{ugc}} = (\langle x \rangle_{\mathsf{gc}}, \mathsf{SK})$.

$\underline{\mathsf{GC.EvalGC}(\langle C \rangle_{\mathsf{ugc}}, \langle x \rangle_{\mathsf{gc}})}$: On input input $\langle C \rangle_{\mathsf{ugc}} = (\langle U \rangle_{\mathsf{gc}}, \mathsf{CT}_1, \ldots, \mathsf{CT}_s)$ and $\langle x \rangle_{\mathsf{ugc}} := (\langle x \rangle_{\mathsf{gc}}, \mathsf{SK})$, do as follows.

1. Using SK, let $\langle C \rangle_{\mathsf{gc}} = (\mathsf{Dec}(\mathsf{SK}, \mathsf{CT}_1), \ldots, \mathsf{Dec}(\mathsf{SK}, \mathsf{CT}_s))$.
2. Return $\mathsf{EvalGC}(\langle U \rangle_{\mathsf{gc}}, \langle C \rangle_{\mathsf{gc}} \circ \langle x \rangle_{\mathsf{gc}})$ where $\circ$ denotes string concatenation.

[8]This is possible by because GC is projective, and therefore these input labels can be generated without knowledge of the input x.

Correctness. Consider a circuit $C \in \mathcal{C}_\lambda$, input $x \in \{0,1\}^\lambda$. Consider a vector of updates $\mathbf{U} \in (\mathcal{U}_\lambda)^q$, where $q(\lambda)$ is a polynomial in λ. Consider the following two processes:

Secure updating process:

1. $(\langle C \rangle_{\mathsf{ugc}}, \mathsf{st}^0_{\mathsf{ugc}}) \leftarrow \mathsf{GrbCkt}\left(1^\lambda, C\right)$.
2. For every $i \in [q]$; $(\langle \mathbf{u}_i \rangle_{\mathsf{ugc}}, \mathsf{st}_i) \leftarrow \mathsf{GenUpd}\left(\mathsf{st}^{i-1}_{\mathsf{ugc}}, \mathbf{u}_i\right)$, where $\mathbf{u}_i$ is the i^{th} entry in $\mathbf{U}$. Each garbled update includes a re-encryption key $\mathsf{RK}_{i,i+1}$ punctured at the index updates by $\mathbf{u}_i$.
3. Let $\langle C_0 \rangle_{\mathsf{ugc}} := \langle C \rangle_{\mathsf{ugc}}$. For every $i \in [q]$, let $\langle C_i \rangle_{\mathsf{ugc}} \leftarrow \mathsf{ApplyUpd}\left(\langle C_{i-1} \rangle_{\mathsf{ugc}},\ \langle \mathbf{u}_i \rangle_{\mathsf{ugc}}\right)$.
4. $\langle x \rangle_{\mathsf{ugc}} \leftarrow \mathsf{GrbInp}(x, \mathsf{st}^q_{\mathsf{ugc}})$, where $\langle x \rangle_{\mathsf{ugc}} = (\langle x \rangle_{\mathsf{gc}}, \mathsf{SK}_q)$.

Insecure updating process:

1. Let $C_0 := C$. For every $i \in [q]$, we have $C_i \leftarrow \mathsf{Upd}(C_{i-1}, \mathbf{u}_i)$. The output is $C_q(x)$.

We need to show that

Claim 3. $\mathsf{EvalGC}\left(\langle C_q \rangle_{\mathsf{ugc}}, \langle x \rangle_{\mathsf{ugc}}\right) = C_q(x)$

Proof. We define notation used in the secure updating process. The state for each $i \in [q]$, $\mathsf{st}^i_{\mathsf{ugc}} = (\mathsf{st}^i, \mathsf{SK}_i, \mathbf{r}^i_{\mathsf{reEnc}})$. For each i, the ith garbled update consists of $\langle \mathbf{u}_i \rangle_{\mathsf{ugc}} = (p_i, \mathsf{CT}^i_{p_i}, \mathsf{RK}^{r^i_{p_i}}_{i,i+1})$, where $p_i \in [s]$ is the bit of C updated in $\mathbf{u}_i$, and $r^i_{p_i}$ is the randomness r_{p_i} from $\mathbf{r}^i_{\mathsf{reEnc}}$ on which $\mathsf{RK}_{i,i+1}$ is punctured, and $\mathsf{CT}^i_{p_1}$ is an encryption of the new garbled wire label for the input p_i encrypted under the key SK_i.

By the correctness of the circuit garbling scheme GC, it suffices to show that $\langle C_q \rangle_{\mathsf{gc}} = (\mathsf{Dec}(\mathsf{SK}_q, \mathsf{CT}^q_1), \ldots, \mathsf{Dec}(\mathsf{SK}_q, \mathsf{CT}^q_s))$ computed during EvalGC are indeed the correct garbled wire labels. CT^q_1 (respectively, for each ciphertext) is computed by a series of re-encryptions using the various punctured re-encryption keys starting from an encryption under some earlier key. This encryption may have been computed using SK_0, or using SK_j if $\mathbf{u}_j$ altered the first bit of $\mathsf{binary}(C)$. In any case, the original ciphertext encrypted the correct garbled label $L_1 := \langle \mathsf{binary}(C)_1 \rangle_{\mathsf{gc}}$. Therefore we must show that the various re-encryptions, and the final decryption with key SK_q, do not introduce any errors.

Let r^*_1 be the randomness used to encrypt the ciphertext under the earlier key. By the correctness of the proxy re-encryption scheme ReEnc and because q is polynomially bounded, $\mathsf{Dec}(\mathsf{SK}_q, \mathsf{CT}^q_1)$ correctly outputs L_1 as long as none of the re-encryption keys were punctured at r^*_1.

With probability at least $1 - (s+q)/(2^\lambda) = 1 - \mathsf{negl}(\lambda)$, all choices of randomness r every sampled for $\mathsf{ReEnc.Enc}$ are unique, where $s+q$ is the total number of fresh encryptions computed. We condition on this event occurring. Therefore, with high probability, none of the re-encryption keys are punctured at r^*_1, completing the proof. □

Efficiency. We lay out different efficiency properties associated with the above scheme.

- *Garbling Time*: The time to compute UGC.$\mathsf{GrbCkt}(1^\lambda, C)$ is equal to $T_{\mathsf{GrbCkt}}(1^\lambda, \mathcal{C}) + |C| \cdot \mathrm{poly}(\lambda)$, where $T_{\mathsf{GrbCkt}}(1^\lambda, \mathcal{C})$ is the time needed to garble the universal circuit for $\mathcal{C}$, along with the input wire labels for C. The second term captures the time to encrypt each of the $|C|$-many garbled labels of C, each encryption taking $\mathrm{poly}(\lambda)$ time to compute. Note that these encryption may be parallelized.

  Using an underlying garbling scheme with GrbCkt in NC^1 (for example, Yao's garbling scheme [Yao86]), and observing that $|U| < |C|^2$, the parallel-time (depth) for computing UGC.$\mathsf{GrbCkt}(1^\lambda, C)$ is $\mathrm{poly}(\lambda, \log |C|)$.

The time to compute UGC.$\mathsf{GrbInp}(\mathsf{st}_{\mathsf{ugc}}, x)$ is $T_{\mathsf{GrbInp}}(\mathsf{st}_{\mathsf{gc}}, x)+\mathrm{poly}(\lambda)$, where $T_{\mathsf{GrbInp}}(\mathsf{st}_{\mathsf{gc}}, x)$ is the time needed to garble the input x. Using Yao's garbling scheme, the time to compute the garbled input is $\mathrm{poly}(\lambda, |x|)$, independent of $|C|$ and the number of updates.

- *Secure Update Generation Time*: The time to generate an update is $\mathrm{poly}(\lambda)$, independent of the size of the circuit C.
- *Secure Update Size*: The size of a garbled update is $|\langle \mathbf{u} \rangle_{\mathsf{ugc}}| = |\mathbf{u}| + \mathrm{poly}(\lambda)$, independent of $|C|$ and the history of updates.
- *State Size*: The size of garbler's state is $|C| \cdot \mathrm{poly}(\lambda)$. Though it grows with $|C|$, it is independent of the history of updates. Using the transformation described in Appendix C, it is possible to remove the dependence on $|C|$ for the state size.
- *Runtime of Update*: The runtime of $\mathsf{ApplyUpd}$ is $\mathrm{poly}(|C|, \lambda)$. It depends on $|C|$, but is independent of the number of updates performed.

IND-Security. The security of UGC follows directly from the security of the proxy re-encryption scheme ReEnc.

In the security game, the adversary sends circuits C^0 and C^1, inputs x^0 and x^1, and sequences of updates $\{(\mathbf{u}_j^0, \mathbf{u}_j^1)\}_{j=1}^q$ such that the indices being updated match (i.e., $\phi_u(\mathbf{u}_j^0) = \phi_u(\mathbf{u}_j^1)$ for all j). In response, the adversary receives a sequence of ciphertexts, punctured re-encryption keys, the secret key SK_q, and a garbled input.

For each $\beta \in \{0, 1\}$, and $i \in [s+\lambda]$ (i.e. $s+\lambda = |\mathsf{binary}(C)|+|x|$), let $L_{i,\beta}$ be the garbled wire label for the ith input bit β of the universal circuit U. That is, $L_{i,\beta} = \mathsf{GC.GrbInputWire}(\mathsf{st}, i)_\beta$. Then for each $i \in [s]$, the adversary receives a series of encryptions of $L_{i,0}$ and $L_{i,1}$, one ciphertext from the initial garbling GrbCkt, and subsequent ciphertexts generated by GenUpd whenever an update $\mathbf{u}$ altered the ith bit of $\mathsf{binary}(C)$.

By construction, whenever an update for the ith bit of $\mathsf{binary}(C)$, it consists of a new encryption of one of $L_{i,0}$ or $L_{i,1}$, along with a re-encryption key that is punctured at the previous ciphertext corresponding to the input bit i. For every i, there is a single "most up-to-date" ciphertext CT_i^* of $L_{i,\mathsf{binary}(C_q^b)_i}$ of the label of the ith bit of the final circuit C_q^b.

It suffices to show that the adversary's view in the real security game is indistinguishable from the view in a modified security game in which all other ciphertexts CT_i' (those that are not the most up-to-date) are in fact generated as $\mathsf{CT}_i' \leftarrow \mathsf{Enc}(\mathsf{SK}, 0)$ as encryptions of 0.

That this is sufficient follows directly from the $\mathsf{PrivIND}_\phi$ security of the garbling scheme GC. The adversary's view in the modified game can be generated given $\langle U \rangle_{\mathsf{gc}} \leftarrow \mathsf{GrbCkt}1^\lambda, U$ and $\langle C_q^b, x^b \rangle_{\mathsf{gc}} \leftarrow \mathsf{GrbInp}(1^\lambda, (C_q^b, x^b))$, where C_q^b is C^b with all updates $\mathbf{u}_j^b$ applied. Because $C^0(x^0) = C^1(x^1)$, the $\mathsf{PrivIND}_\phi$ security of GC implies that the bit b is computationally hidden.

Indistinguishability of these two games follows from the security of ReEnc. Observe that for any CT_i' that is not the most up-to-date, there exists a later update that modifies the bit i. By construction, when that update is issued, the corresponding re-encryption key will be punctured at the ciphertext CT_i'. Indistinguishability maps directly to the security of ReEnc. One sequence of messages are the real sequence of labels used to generate the encryptions of CT_i^* and CT_i' for each i, while the second is the sequence of labels where all but the most up-to-date are replaced with 0. Similarly, the sequence of punctured key requests $(p_1, \ldots, p_q)$ and the key-indices $(k_1, \ldots, k_\ell)$ (where $\ell = s + q$) as defined in Definition 22 correspond to the sequence re-encryption keys and fresh encryptions generated by GenUpd.

6 Updatable Cryptography

We define the notion of updatability in the context of several cryptographic primitives. This is done by first considering primitives based on IND-security notion and then we also consider simulation based primitives.

In Section 6.1 view the primitives based on IND-security via the lens of *circuit compilers* and to consider the context of updatability, we define the notion of *dynamic* circuit compilers. We then remark how dynamic circuit compilers can be used to instantiate several IND-security based cryptographic primitives.

We then move on to simulation based security notions. We first define the notion of updatable non interactive zero knowledge in Section 6.2. We then define the notion of updatable multiparty computation in Section 6.3.

6.1 Dynamic Circuit Compilers

We introduce the notion of circuit compilers below. It consists of the algorithms $\mathsf{CC} = (\mathsf{Gen}, \mathsf{Compile}, \mathsf{Encode}, \mathsf{Decode})$. Its associated with a class of circuits $\mathcal{C}$.

- **Generation of Parameters,** $\mathsf{Gen}(1^\lambda)$: On input security parameter λ, outputs parameters $(\mathsf{cktSK}, \mathsf{inpSK})$ and public parameters pp.

- **Circuit compilation,** $\mathsf{Compile}(\mathsf{cktSK}, C)$: On input secret parameters cktSK and circuit C, it outputs an encoding of circuit $\langle C \rangle$ and state st.

- **Input encoding,** $\mathsf{Encode}(\mathsf{inpSK}, x)$: On input parameters inpSK and input x, it outputs an encoding of input $\langle x \rangle$.

- **Evaluation,** $\mathsf{Eval}(\langle C \rangle, \langle x \rangle)$: On input encodings $\langle C \rangle$ and $\langle x \rangle$, it outputs the decoded value α.

There are two properties associated with a circuit compiler - correctness and security. We consider sequential updating process. However these properties can also be studied for other updating processes.

Correctness. Consider a circuit C and an input x. We require that the evaluation of encoding $\langle C \rangle$ on $\langle x \rangle$ yields $C(x)$.

ρ-Security. The security property is associated with ρ-admissibility property. We consider indistinguishability security notion but the same can also be defined with respect to a simulation style definition.

We describe the game between the challenger and adversary below.

1. The adversary sends circuit pair (C^0, C^1), circuits $(C_1, \ldots, C_{\ell_f})$ and input pairs $(x_1^0, x_1^1), \ldots, (x_{\ell_{\mathsf{inp}}}^0, x_{\ell_{\mathsf{inp}}}^1)$. Denote this information as aux.

2. Challenger first executes $\rho(aux)$. If the output is 0, it aborts. Else, it computes the parameters pp and $(\mathsf{cktSK}, \mathsf{inpSK})$. It then picks a bit b at random. It computes $(\langle C^b \rangle, \mathsf{st}_0) \leftarrow \mathsf{Compile}(\mathsf{cktSK}, C^b)$. It computes $(\langle C_i \rangle, \mathsf{st}_{i,0}) \leftarrow \mathsf{Compile}(\mathsf{cktSK}, C_i)$ for $i \in [\ell_f]$. It computes $\langle x_k^b \rangle \leftarrow \mathsf{Encode}(\mathsf{inpSK}, x_k^b)$ for $k \in [\ell_{\mathsf{inp}}]$. It sends the following the adversary:
$$\left(\langle C^b \rangle,\ \{\langle C_i \rangle\}_{i \in [\ell_f]},\ \{\langle x_k^b \rangle\}_{k \in [\ell_{\mathsf{inp}}]}\right)$$

3. Adversary outputs a bit b'.

We say that the circuit compiler scheme is secure if the following holds.

Definition 25 (Security). *A circuit compiler scheme* CC *is secure if the probability that the adversary outputs* $b' = b$ *in the above experiment with probability negligibly close to* $1/2$.

Instantiations. Several advanced cryptographic primitives can be seen in the form of circuit compilers. We give a couple of examples.

- *Attribute Based Encryption (ABE)*: In the case of ABE; $\mathsf{Gen} = \mathsf{Setup}$ (setup), $\mathsf{Compile} = \mathsf{KeyGen}$ (key generation), $\mathsf{Encode} = \mathsf{Enc}$ (encryption), $\mathsf{Eval} = \mathsf{Dec}$ (decryption). Furthermore every circuit $C \in \mathcal{C}$ associated with this scheme is of the form: $C(x = (attr, m)) := m$ if and only if $C'(attr) = 1$ for some circuit C' hardwired in C. Depending on whether the ABE scheme we are considering is public key scheme or not, we can assign inpSK to be either public parameters or secret key parameters.
  In terms of security, ρ interprets x_i^b as $(attr_i^b, m_i^b)$ and checks if $attr_i^0 = attr_i^1$. It also checks if $C^0 = C^1$. If either of the checks fail, ρ outputs 0.
- *Non Interactive Witness Indistinguishable Proof Systems (WI)*: Consider a non interactive witness indistinguishable system $\mathsf{WI} = (P, V)$ associated with a NP relation R. We denote by P the circuit representing the prover. Suppose x is the NP instance and w is the witness associated with x. Here, $\mathsf{Gen} = \mathsf{null}$(meaning that the algorithm Gen is not defined), $\mathsf{Compile} = P(x, w; r)$ (prover circuit with instance, witness, randomness hardwired), $\mathsf{Encode} = \mathsf{null}$, $\mathsf{Eval} = V$ (verifier circuit).
- *Indistinguishability Obfuscation (iO)*: Here, $\mathsf{Gen} = \mathsf{null}$ (meaning that the algorithm Gen is not defined), $\mathsf{Compile} = \mathsf{Obf}$ (obfuscate), $\mathsf{Eval} = \mathsf{iOEval}$ (evaluation of obfuscation). In terms of security, ρ checks if $C^0 \equiv C^1$ and $C_i = \perp$ for $i \in [\ell_f]$, $x_k = \perp$ for $k \in [\ell_{\mathsf{inp}}]$. If the check fails it outputs 0, else it outputs 1.

Dynamic Circuit Compilers. The notion of dynamic circuit compilers additionally have the algorithms GenUpd (generation of secure update) and $\mathsf{ApplyUpd}$ (apply secure update) associated with it. It consists of the algorithms $\mathsf{DCC} = (\mathsf{Gen}, \mathsf{Compile}, \mathsf{Encode}, \mathsf{GenUpd}, \mathsf{ApplyUpd}, \mathsf{Decode})$. Its associated with a class of circuits $\mathcal{C}$.

- **Generation of Updates,** $\mathsf{GenUpd}(\mathsf{cktSK}, \mathsf{st}, \mathbf{u})$: On input secret parameters cktSK, state st, update $\mathbf{u}$, it outputs an encoding of update $\langle \mathbf{u} \rangle$ and an updated state st'.
- **Apply Update,** $\mathsf{ApplyUpd}(\mathsf{pp}, \langle C \rangle, \langle \mathbf{u} \rangle)$: On input public parameters pp, circuit encoding $\langle C \rangle$, secure update $\langle \mathbf{u} \rangle$, it outputs an updated circuit encoding $\langle C' \rangle$.

Correctness. Consider a circuit $C = C_0$, input x and a sequence of updates $\mathbf{u}_1, \ldots, \mathbf{u}_q$. Let C_i be the circuit by updating C_{i-1} using update $\mathbf{u}_i$. We require that the evaluation of encoding $\langle C_i \rangle$ on $\langle x \rangle$ yields $C_i(x)$, where $\langle C_i \rangle$ is obtained by updating $\langle C_{i-1} \rangle$ using $\langle \mathbf{u}_i \rangle$ (encoding of $\mathbf{u}_i$) and is associated with a new secret key $\mathsf{CC.cktSK}_i$. The encoding $\langle x \rangle$ is computed using the new secret key $\mathsf{CC.cktSK}_i$.

Remark 1. *We emphasize that once the circuit is updated, the secret key associated with the compiled circuit could potentially change. Hence, we require that correctness to be satisfied only for input encodings created using the new secret key.*

ρ-Security. The security property is associated with ρ-admissibility property. We consider indistinguishability security notion but the same can also be defined with respect to a simulation style definition.

We describe the game between the challenger and adversary below.

1. The adversary sends circuit pair (C^0, C^1), circuits $(C_1, \ldots, C_{\ell_f})$, input pairs $(x_1^0, x_1^1), \ldots, (x_{\ell_{\mathsf{inp}}}^0, x_{\ell_{\mathsf{inp}}}^1)$ and update pairs $(\mathbf{u}_1^0, \mathbf{u}_1^1) \ldots, (\mathbf{u}_q^0, \mathbf{u}_q^1)$. Denote this information as aux.

2. Challenger first executes $\rho(aux)$. If the output is 0, it aborts. Else, it computes the parameters pp and $(\mathsf{cktSK}, \mathsf{inpSK})$. It then picks a bit b at random. It computes $(\langle C^b \rangle, \mathsf{st}_0) \leftarrow \mathsf{Compile}(\mathsf{cktSK}, C^b)$. It computes $(\langle C_i \rangle, \mathsf{st}_{i,0}) \leftarrow \mathsf{Compile}(\mathsf{cktSK}, C_i)$ for $i \in [\ell_f]$. It computes $\langle \mathbf{u}_{j,b} \rangle \leftarrow \mathsf{GenUpd}(\mathsf{cktSK}, \mathsf{st}_{j-1}, \mathbf{u}_j)$ for $j \in [q]$. It computes $\langle x_k^b \rangle \leftarrow \mathsf{Encode}(\mathsf{inpSK}, x_k^b)$ for $k \in [\ell_{\mathsf{inp}}]$. It sends the following the adversary:
$$\left(\langle C^b \rangle,\ \{\langle C_i \rangle\}_{i \in [\ell_f]}, \{\langle \mathbf{u}_{j,b} \rangle\}_{j \in [q]},\ \{\langle x_k^b \rangle\}_{k \in [\ell_{\mathsf{inp}}]}\right)$$

3. Adversary outputs a bit b'.

We say that the dynamic circuit compiler scheme is secure if the following holds.

Definition 26 (Security)**.** *A dynamic circuit compiler scheme* DCC *is secure if the probability that the adversary outputs $b' = b$ in the above experiment with probability negligibly close to* $1/2$.

Instantiations. We can consider the updatability versions of several cryptographic primitives via the lens of dynamic circuit compilers. For example, we can consider the notion of updatable ABE: In updatable ABE, an attribute key associated with a function, say sk_C, issued can sequentially updated.

Remark 2. *Since we consider dynamic circuit compilers, where only one circuit can be updated in the security game, this correspondingly leads to defining updatable ABE scheme where only one attribute key is updated in the security game. However, for other primitives such as* updatable *indistinguishability obfuscation and* updatable *non interactive witness indistinguishable systems, it suffices to just consider dynamic circuit compilers where only one circuit is updated.*

We can also consider the more general setting where multiple circuits are updated simultaneously using the same sequence of updates. Such a setting was studied in the context of indistinguishability obfuscation [AJS15b]. We do not deal with this setting in this work.

Construction of Dynamic Circuit Compilers. We sketch a construction of dynamic circuit compilers using output-compact updatable randomized encodings. We start with circuit compilers, not necessarily supporting updatability property, and show how to transform it into a dynamic circuit compilers scheme. Let the circuit compilers scheme be denoted by $\mathsf{CC} = (\mathsf{CC.Gen}, \mathsf{CC.Compile}, \mathsf{CC.Encode}, \mathsf{CC.Eval})$. We construct $\mathsf{DCC} = (\mathsf{Gen}, \mathsf{Compile}, \mathsf{Encode}, \mathsf{GenUpd}, \mathsf{ApplyUpd}, \mathsf{Eval})$ as follows.

- $\mathsf{Gen}(1^\lambda)$ executes $\mathsf{CC.Gen}(1^\lambda)$ and also the setup of updatable randomized encoding scheme. The parameters output by Gen is the joint parameters output by both $\mathsf{CC.Gen}$ and the setup of URE.

- $\mathsf{CC.Compile}$, on input secret parameters and circuit C, executes an (updatable) randomized encoding of $\mathsf{Compile}(\cdot, \cdot; \cdot)$ and input $(\mathsf{CC.cktSK}, C, r)$, where $\mathsf{CC.cktSK}$ is the secret parameters output by CC and r is the randomness used in the circuit compilation process.

- GenUpd on input secret parameters and update $\mathbf{u}$, it generates a secure update $\langle (\mathbf{u}, \mathsf{CC.cktSK}', r') \rangle$, where $\mathsf{CC.cktSK}'$ is the new secret key and r' is a new randomness to be used in the circuit compilation algorithm, computed with respect to updatable randomized encodings scheme.

- $\mathsf{ApplyUpd}$ updates the randomized encoding, part of the compiled circuit, using the secure update. It outputs the updated randomized encoding.

- Encode is identical to $\mathsf{CC.Encode}$.
- Eval first decodes the randomized encoding to get a compiled circuit, computed with respect to CC. It then executes the evaluation algorithm $\mathsf{CC.Eval}$.

We omit the proof of correctness argument since it follows directly from the proof of correctness of the underlying circuit compilers and updatable randomized encodings scheme. We sketch the proof of security below.

In terms of efficiency, the output-compactness property of the underlying URE scheme guarantees that the size of the updates are small. Note that if not for the output-compactness property, the size of the updates could be proportional to the output length of the circuit which is proportional to the size of the circuit being compiled.

Security: The main steps in the security argument are as follows: Let (C_0, C_1) be the circuit pair, $(x_1^0, x_1^1), \ldots, (x_{\ell_{\mathsf{inp}}}^0, x_{\ell_{\mathsf{inp}}}^1)$ be the input pairs and update pairs $(\mathbf{u}_1^0, \mathbf{u}_1^1) \ldots, (\mathbf{u}_q^0, \mathbf{u}_q^1)$.

1. In the first step, instead of computing the randomized encoding of circuit compile algorithm $\mathsf{Compile}$ on just C_0, it does the following: consider the circuit G which is computed as follows:

   $$G(\mathit{temp} = 0, \mathsf{CC.cktSK}, C^0, r, C^1, r') \text{ outputs } \mathsf{Compile}(\mathsf{CC.cktSK}, C^0; r).$$

   Furthermore, the secure updates are computed as an encoding of $(i, \mathbf{u}_i^0, r_i^0, \mathbf{u}_i^1, r_i^1, \mathsf{CC.cktSK}_i)$, where $(\mathbf{u}_i^0, r_i^0)$ (resp., $(\mathbf{u}_i^1, r_i^1)$) will be used to update C_{i-1}^0 to C_i^0 (resp., C_{i-1}^1 to C_i^1). The value i, as part of update encoding, is used to update the value temp from 0 to i. Finally, $\mathsf{CC.cktSK}_i$ is the new secret key.

2. In a sequence of steps, the code of G is switched from computing a compiled circuit of C^0 to computing a compiled circuit of C^1. This is done in a sequence of steps: in the j^{th} step, for $j \in [q+1]$, G is of the following form:

   $$G(\mathit{temp} = 0, \mathsf{CC.cktSK}, C^0, r, C^1, r') := \begin{cases} \mathsf{Compile}(\mathsf{CC.cktSK}, C^0; r) & \text{if } \mathit{temp} \geq j, \\ \mathsf{Compile}(\mathsf{CC.cktSK}, C^1; r) & \text{otherwise} \end{cases}$$

   The j^{th} step is switched to $(j+1)^{th}$ step is done in the following steps: a third branch is introduced in G, for the case when $\mathit{temp} = j$, a hardwired value V is output. For all values of temp, G remains unchanged as in j^{th} step. The value V is set to be $\mathsf{Compile}(\mathsf{CC.cktSK}_j, C_j^0; r_j^0)$, where C_j^0 is the j^{th} updated circuit. Furthermore, in the j^{th} update, the secret key $\mathsf{CC.cktSK}_j$ and randomness (r_j^0, r_j^1) is removed from the description of updates. Now, we invoke the security of underlying dynamic circuit compiler scheme (since $\mathsf{CC.cktSK}_j$ is "removed" from the system) to switch from $\mathsf{Compile}(\mathsf{CC.cktSK}_j, C_j^0; r_j^0)$ to $\mathsf{Compile}(\mathsf{CC.cktSK}_j, C_j^1; r_j^1)$. Once this is done, we can switch the description of G from having the instruction "$\mathit{temp} \geq j$" to "$\mathit{temp} \geq j+1$". We invoke the security of URE to make this change.

3. In the $(q+1)^{th}$ step, G outputs a compiled circuit of C^1 for every value of temp. Thus, C^0 and updates $\{\mathbf{u}_i^0\}$ can now be removed from the system.

6.2 Updatable Non-Interactive Zero Knowledge

We start by giving the syntax and formal definition.

Syntax. Let R be an efficiently computable relation that consists of pairs (x, w), where x is called the statement and w is the witness. Let L denote the language consisting of statements in R. We say that the relation R is $(\mathsf{Upd}, \mathcal{U})$-updatable if for any $(x, w) \in R$ and any update string $\mathbf{u} \in \mathcal{U}$, $\mathsf{Upd}(x, w, \mathbf{u}) = (x', w')$ s.t. $(x', w') \in R$.

An updatable non-interactive zero-knowledge (UNIZK) proof system for a language L with a $(\mathsf{Upd}, \mathcal{U})$-updatable relation R consists of a setup algorithm CRSGen, a prover algorithm Prove and a verifier algorithm Verify similar to a standard NIZK proof system. However, it also comes equipped with two additional algorithms, namely, GenUpd and ApplyUpd. We formally describe the algorithms below:

- **Setup,** $\mathsf{crs} \leftarrow \mathsf{CRSGen}(1^\lambda)$: On input a security parameter λ, it outputs a common reference string crs.
- **Prove,** $(\pi, \mathsf{st}) \leftarrow \mathsf{Prove}(\mathsf{crs}, x, w)$: On input a common reference string crs and a statement x along with a witness w, it first checks whether $(x, w) \in R$; if so, it produces a proof string π along with a state st, else it outputs `fail`.
- **Generate Update,** $\langle \mathbf{u} \rangle \leftarrow \mathsf{GenUpd}(\mathbf{u}, \mathsf{st})$: On input an update $\mathbf{u} \in \mathcal{U}$ and state st, it outputs an update encoding $\langle \mathbf{u} \rangle$ and a new state st'.
- **Apply Update,** $\pi' \leftarrow \mathsf{ApplyUpd}(\mathsf{crs}, \pi, \langle \mathbf{u} \rangle)$: On input a common reference string crs, a proof string π and an update encoding $\langle \mathbf{u} \rangle$, it outputs an updated proof string π'.
- **Verify,** $b \leftarrow \mathsf{Verify}(\mathsf{crs}, x, \pi)$: On input a common reference string crs, a statement x and a proof string π, it outputs $b = 1$ if the proof is valid, and $b = 0$ otherwise.

Definition 27 (Updatable NIZKs)**.** *An updatable non-interactive zero-knowledge (*UNIZK*) proof system for a language L with a* PPT *relation R and update family $\mathcal{U}$ with updating algorithm* Upd *is a tuple of* PPT *algorithms* (CRSGen, Prove, GenUpd, ApplyUpd, Verify) *such that the following properties hold:*

- **Efficiency:** *For any update $\mathbf{u} \in \mathcal{U}$, the running time of* $\mathsf{GenUpd}(\mathbf{u}, \mathsf{st})$ *is $p(\lambda, |\mathbf{u}|)$, where p is an a priori fixed polynomial. This implies that the size of the resultant update encoding* $\langle \mathbf{u} \rangle \leftarrow \mathsf{GenUpd}(\mathbf{u}, \mathsf{st})$ *is also a fixed polynomial in λ and $|\mathbf{u}|$.*
- **Completeness:** *For every $(x_0, w_0) \in R$ and every sequence of updates $\mathbf{u}_1 \ldots, \mathbf{u}_q \in \mathcal{U}$, it holds that for every $0 \leq i \leq q$:*

$$\Pr[\mathsf{Verify}(\mathsf{crs}, x_i, \pi_i) = 1] = 1$$

  where $\mathsf{crs} \leftarrow \mathsf{CRSGen}(1^\lambda)$, $(\pi_0, \mathsf{st}_0) \leftarrow \mathsf{Prove}(\mathsf{crs}, x_0, w_0)$, $(\langle \mathbf{u}_i \rangle, \mathsf{st}_i) \leftarrow \mathsf{GenUpd}(\mathbf{u}_i, \mathsf{st}_{i-1})$, $\pi_i \leftarrow \mathsf{ApplyUpd}(\pi_{i-1}, \langle \mathbf{u}_i \rangle)$ *and the probability is taken over the coins of* CRSGen*,* Prove*,* GenUpd*,* ApplyUpd *and* Verify*.*
- **Soundness against Sequential Updates:** *For every adversary $\mathcal{A}$, there exists a negligible function* $\mathsf{negl}(\lambda)$ *s.t.*

$$\Pr[1 \leftarrow \mathsf{Expt}_{\mathcal{A}}^{\mathsf{pu\text{-}sound}}(1^\lambda)] \leq \mathsf{negl}(\lambda)$$

  where, $\mathsf{Expt}_{\mathcal{A}}^{\mathsf{pu\text{-}sound}}(1^\lambda)$ *is defined as follows:*

  1. Ch *computes* $\mathsf{crs} \leftarrow \mathsf{CRSGen}(1^\lambda)$ *and sends* crs *to $\mathcal{A}$.*
  2. *$\mathcal{A}$ outputs* $(x_0, \pi_0, \{x_i, \langle \mathbf{u} \rangle_i\}_{i=1}^{q})$ *to* Ch*.*
  3. *The output of the experiment is* 1 *if:* either $\mathsf{Verify}(\mathsf{crs}, x_0, \pi_0) = 1 \wedge x_0 \notin L$, or *there exists* $1 \leq i \leq q$ *s.t.* $\mathsf{Verify}(\mathsf{crs}, x_i, \pi_i) = 1 \wedge x_i \notin L$ *where* $\pi_i \leftarrow \mathsf{ApplyUpd}(\pi_{i-1}, \langle \mathbf{u}_i \rangle)$*. Otherwise, the output is* 0*.*

If the soundness property only holds against PPT *adversaries, then we call it an* argument *system.*

- **Zero Knowledge against Sequential Updates:** *There exists a PPT simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3)$ s.t. for every $(x_0, w_0) \in R$, every auxiliary input $z \in \{0,1\}^*$ and every sequence of updates $\mathbf{u}_1, \ldots, \mathbf{u}_q \in \mathcal{U}$, it holds that*

$$\mathsf{IDEAL}\Big(1^\lambda, x_0, w_0, z, \{\mathbf{u}_i\}_{i=1}^q\Big) \approx_c \mathsf{REAL}\Big(1^\lambda, x_0, w_0, z, \{\mathbf{u}_i\}_{i=1}^q\Big)$$

where,

$\underline{\mathsf{IDEAL}\ (1^\lambda, x_0, w_0, z, \{\mathbf{u}_i\}_{i=1}^q)}$:	$\underline{\mathsf{REAL}\ (1^\lambda, x_0, w_0, z, \{\mathbf{u}_i\}_{i=1}^q)}$:
1. $(\mathsf{crs}, \mathsf{st}_1) \leftarrow \mathcal{S}_1(1^\lambda)$	*1.* $\mathsf{crs} \leftarrow \mathsf{CRSGen}(1^\lambda)$
2. $(\pi_0, \mathsf{st}_2) \leftarrow \mathcal{S}_2(x_0, \mathsf{st}_1)$	*2.* $(\pi_0, \mathsf{st}_0) \leftarrow \mathsf{Prove}(\mathsf{crs}, x_0, w_0)$
3. $\forall i \in [q]$, $(x_i, w_i) \leftarrow \mathsf{Upd}(x_0, w_0, \mathbf{u}_i)$	*3.* $\forall i \in [q]$, $\langle \mathbf{u}_i \rangle \leftarrow \mathsf{GenUpd}\ (\mathbf{u}_i, \mathsf{st}_{i-1})$
4. $\forall i \in [q]$, $\langle \mathbf{u}_i \rangle \leftarrow \mathcal{S}_3(\mathsf{st}_2, 1^{\lvert \mathbf{u}_i \rvert}, x_i)$.	
Output $(x_0, z, \pi_0, \{\langle \mathbf{u} \rangle_i\}_{i=1}^p)$	*Output* $(x_0, z, \pi_0, \{\langle \mathbf{u}_i \rangle\}_{i=1}^q)$

6.2.1 Construction of UNIZK

In this section, we construct a UNIZK proof system for **NP**. Let L be any language in **NP** with a $(\mathsf{Upd}, \mathcal{U})$-updatable PPT relation R. We construct a UNIZK proof system $(\mathsf{CRSGen}, \mathsf{Prove}, \mathsf{GenUpd}, \mathsf{ApplyUpd}, \mathsf{Verify})$ for L.

Notation. Let $R[x, w]$ denote a hardwired circuit corresponding to R with inputs (x, w). Let $R[X, W]$ denote the corresponding hardwired circuit family. We will use the following ingredients in our construction:

1. A stateless[9] URE scheme $(\mathsf{URE.Encode}, \mathsf{URE.GenUpd}, \mathsf{URE.ApplyUpd}, \mathsf{URE.Decode})$ for an $(\mathsf{Upd}_{\mathsf{ure}}, \mathcal{U}_{\mathsf{ure}})$-updatable class of hardwired circuits $R[X, W]$ where $\mathcal{U}_{\mathsf{ure}} = \mathcal{U}$ and $\mathsf{Upd}_{\mathsf{ure}}$ is s.t. for any $\mathbf{u} \in \mathcal{U}_{\mathsf{ure}}$, $\mathsf{Upd}_{\mathsf{ure}}(R[x, w], \mathbf{u}) = R[x', w']$ where $(x', w') \leftarrow \mathsf{Upd}(x, w, \mathbf{u})$.
2. A non-interactive perfectly binding commitment scheme Com. Such a scheme can be based on the existence of injective one-way functions.[10]
3. A NIZK proof of knowledge (NIZKPOK) system $(\mathsf{NIZK.CRSGen}, \mathsf{NIZK.Prove}, \mathsf{NIZK.Verify})$ for **NP**.

Construction. We now describe the construction of UNIZK.

$\underline{\mathsf{CRSGen}(1^\lambda)}$: Sample a common reference string $\mathsf{crs}_{\mathsf{nizk}} \leftarrow \mathsf{NIZK.CRSGen}(1^\lambda)$ for the NIZKPOK system. Output $\mathsf{crs} = \mathsf{crs}_{\mathsf{nizk}}$.

$\underline{\mathsf{Prove}(x, w)}$: Perform the following sequence of steps:

- Sample a random string r_{ure} and compute $\langle R[x, w] \rangle_{\mathsf{ure}} \leftarrow \mathsf{URE.Encode}\left(1^\lambda, R, (x, w)\,; r_{\mathsf{ure}}\right)$ using randomness r_{ure}.
- Sample a random string r_{com} and compute a commitment $C \leftarrow \mathsf{Com}(r_{\mathsf{ure}}; r_{\mathsf{com}})$ to r_{ure} using randomness r_{com}.

[9] Our construction also works if we start with a stateful URE scheme. For simplicity of exposition, however, we present our construction using a stateless URE scheme. As a result, we actually construct a stateless UNIZK scheme.

[10] We can, in fact, use a two round statistically binding commitment scheme that can be based on standard one-way functions. For simplicity of exposition, however, we present our construction using non-interactive commitments.

- Compute a proof $\pi_{\mathsf{nizk}} \leftarrow \mathsf{NIZK.Prove}(x_{\mathsf{nizk}}, w_{\mathsf{nizk}})$ for the statement $x_{\mathsf{nizk}} = (x, \langle R[x,w]\rangle_{\mathsf{ure}}, C)$ using witness $w_{\mathsf{nizk}} = (w, r_{\mathsf{ure}}, r_{\mathsf{com}})$ where $(w', r'_{\mathsf{ure}}, r'_{\mathsf{com}})$ is a valid witness for x_{nizk} iff all of the following hold:
  - $\langle R[x,w]\rangle_{\mathsf{ure}} \leftarrow \mathsf{URE.Encode}\left(1^\lambda, R, (x, w')\,; r'_{\mathsf{ure}}\right)$
  - $C \leftarrow \mathsf{Com}(r'_{\mathsf{ure}}; r'_{\mathsf{com}})$

Finally, set $\mathsf{st} = (r_{\mathsf{ure}}, C, r_{\mathsf{com}})$ and output $\pi = (\langle R[x,w]\rangle_{\mathsf{ure}}, C, \pi_{\mathsf{nizk}})$.

$\underline{\mathsf{GenUpd}(\mathbf{u}, \mathsf{st})}$: Perform the following sequence of steps:

- Parse $\mathsf{st} = (r_{\mathsf{ure}}, C, r_{\mathsf{com}})$.
- Sample a random string r_{upd} and compute $\langle \mathbf{u}\rangle_{\mathsf{ure}} \leftarrow \mathsf{URE.GenUpd}(\mathbf{u}, r_{\mathsf{ure}}; r_{\mathsf{upd}})$ using randomness r_{upd}.
- Compute a proof $\pi_{\mathsf{nizk}} \leftarrow \mathsf{NIZK.Prove}(x_{\mathsf{nizk}}, w_{\mathsf{nizk}})$ for the statement $x_{\mathsf{nizk}} = (\langle \mathbf{u}\rangle_{\mathsf{ure}}, C)$ using witness $w_{\mathsf{nizk}} = (\mathbf{u}, r_{\mathsf{upd}}, r_{\mathsf{ure}}, r_{\mathsf{com}})$ where $(\mathbf{u}', r'_{\mathsf{upd}}, r'_{\mathsf{ure}}, r'_{\mathsf{com}})$ is a valid witness for x_{nizk} iff all of the following hold:
  - $\langle \mathbf{u}\rangle_{\mathsf{ure}} \leftarrow \mathsf{URE.GenUpd}\left(\mathbf{u}', r'_{\mathsf{ure}}; r'_{\mathsf{upd}}\right)$
  - $C \leftarrow \mathsf{Com}(r'_{\mathsf{ure}}; r'_{\mathsf{com}})$

Output $\langle \mathbf{u}\rangle = (\langle \mathbf{u}\rangle_{\mathsf{ure}}, C, \pi_{\mathsf{nizk}})$.[11]

$\underline{\mathsf{ApplyUpd}(\mathsf{crs}, \pi, \langle \mathbf{u}\rangle)}$: Perform the following steps:

- Parse $\langle \mathbf{u}\rangle = (\langle \mathbf{u}\rangle_{\mathsf{ure}}, C, \pi^{\mathsf{upd}}_{\mathsf{nizk}})$ and $\mathsf{crs} = \mathsf{crs}_{\mathsf{nizk}}$. Let $x^{\mathsf{upd}}_{\mathsf{nizk}} = (\langle \mathbf{u}\rangle_{\mathsf{ure}}, C)$. If $\mathsf{NIZK.Verify}(\mathsf{crs}_{\mathsf{nizk}}, x^{\mathsf{upd}}_{\mathsf{nizk}}, \pi^{\mathsf{upd}}_{\mathsf{nizk}}) = 0$, then output $\perp$.
- If π is a level-0 proof, then parse $\pi = (\langle R[x,w]\rangle_{\mathsf{ure}}, C, \pi_{\mathsf{nizk}})$. Else, parse $\pi = \langle R[x,w]\rangle_{\mathsf{ure}}$.
- Compute $\langle R[x', w']\rangle_{\mathsf{ure}} \leftarrow \mathsf{URE.ApplyUpd}(\langle R[x,w]\rangle_{\mathsf{ure}}, \langle \mathbf{u}\rangle_{\mathsf{ure}})$.
- Output $\langle R[x', w']\rangle_{\mathsf{ure}}$.

$\underline{\mathsf{Verify}(\mathsf{crs}, x, \pi)}$: Perform the following steps:

- Parse $\mathsf{crs} = \mathsf{crs}_{\mathsf{nizk}}$.
- If π is a level-0 proof, then parse $\pi = (\langle R[x,w]\rangle_{\mathsf{ure}}, C, \pi_{\mathsf{nizk}})$. Let $x_{\mathsf{nizk}} = (x, \langle R[x,w]\rangle_{\mathsf{ure}}, C)$. Output $\mathsf{NIZK.Verify}(\mathsf{crs}_{\mathsf{nizk}}, x_{\mathsf{nizk}}, \pi_{\mathsf{nizk}}) \wedge \mathsf{URE.Decode}(\langle R[x,w]\rangle_{\mathsf{ure}})$.
- Else, parse $\pi = \langle R[x,w]\rangle_{\mathsf{ure}}$. Output $\mathsf{URE.Decode}(\langle R[x,w]\rangle_{\mathsf{ure}})$.

Proof Sketch. We first argue the efficiency property. Recall that the computation of an update encoding for UNIZK involves two main steps: first, we compute an update encoding for the underlying URE scheme. Next, we compute a fresh proof string for the underlying NIZKPOK system to prove that the URE update encoding was computed honestly. From the efficiency property of the URE, it follows that the first step only requires time polynomial in the update size and the security parameter. Further, it follows from the standard efficiency of NIZKPOKs that the second step also only requires time polynomial in the size of the update and the security parameter. Putting the above together, we have that GenUpd satisfies the efficiency requirement.

Correctness of the above construction is easy to verify. In order to argue soundness, we leverage the proof of knowledge property of the underlying NIZKPOK. Let us assume that the construction is not sound, i.e., there exists an efficient adversary for the soundness security game that outputs $(x_0, \pi_0, \{x_i, \langle \mathbf{u}_i\rangle\}_{i=1}^q)$ for some q s.t. all the (updated) proof strings are

[11]Note that it is not really necessary for the update encoding to include C since the verifier, who runs the ApplyUpd and Verify algorithms, can remember C. We have added C to the update encoding for clarity.

accepted by the honest verification algorithm, yet at least one of the (updated) statements is false. We obtain a contradiction as follows: for every i, let $\langle \mathbf{u}_i \rangle = (\langle \mathbf{u}_i \rangle_{\mathsf{ure}}, \pi^i_{\mathsf{nizk}})$ and let x^i_{nizk} be the statement corresponding to π^i_{nizk}. Then, starting with $i = q$ and proceeding backwards, we apply the NIZKPOK extractor on each π^i_{nizk} to extract a valid witness w^i_{nizk} for x^i_{nizk}. From the description of the scheme, it follows that w^i_{nizk} consists of an update $\mathbf{u}_i$ as well as randomness that can be used to verify that $\mathbf{u}_i$ was indeed used to compute the update encoding honestly. For $i = 0$, the extractor also returns the witness w_0 for the original statement x_0. Putting all of this together, we can recover a witness w_i for every updated statement x_i, which leads to a contradiction.

Finally, we argue the zero-knowledge property. We establish this via a simple hybrid argument: the first hybrid H_0 corresponds to the real world experiment. In the next hybrid H_1, we first simulate $\mathsf{crs}_{\mathsf{nizk}}$ and all the proof strings π^i_{nizk} that are part of the original proof π_0 and the update encodings $\mathbf{u}_i$. Next, in hybrid H_2, we switch the commitment C to be commitment of all zeros. Finally, in H_3, we use the simulator of URE to simulate the URE encoding $\langle R[x, w] \rangle_{\mathsf{ure}}$ in π_0 as well as the URE update encodings $\langle \mathbf{u} \rangle^i_{\mathsf{ure}}$ in every update $\langle \mathbf{u}_i \rangle$. Note that this experiment corresponds to the ideal world.

The indistinguishability of H_0 and H_1 follows from the zero-knowledge property of the underlying NIZKPOK. The indistinguishability of H_1 and H_2 follows from the hiding property of the commitment scheme Com. Finally, the indistinguishability of H_2 and H_3 follows from the security of the URE scheme. This finishes the proof sketch of our construction.

6.3 Updatable Multiparty Computation

We consider the setting of n parties $\mathcal{P} = \{P_1, ..., P_n\}$ who wish to jointly compute any PPT function over their private inputs. We are interested in the scenario where after performing a computation of any function f over an input vector $\vec{x} = x_1, \ldots, x_n$, the parties wish to perform another computation over an *updated* function f' and input vector $\vec{x}'$. Note that if the parties were to simply use a standard MPC protocol to perform a fresh computation over the updated function and input vector, then the communication complexity of this computation will depend on $|f'|$ and $|\vec{x}'|$. We instead consider the scenario where after performing the initial computation of f over $\vec{x}$, the parties can run a less-expensive *update phase* whose communication complexity only depends on the description size of the update and not on the size of the updated function and input vector. We refer to a protocol that achieves this efficiency property as *updatable MPC* (UMPC).

We consider the setting of multiple updates where the parties can perform multiple updated computations in a *sequential* manner. We now proceed to formally define UMPC with sequential updating using the real/ideal paradigm.

Notation. We study the notion of UMPC with respect to a class of updatable hardwired circuits. Below, we first extend the notation for updatable hardwired circuits (as described in Section 3) to meet our requirements for UMPC. We consider two changes: (a) First, we consider computation over n different inputs, as opposed to a single input. (b) Second, we consider updates $\mathbf{u}$ of the form $\mathbf{u} = u_1, \ldots, u_n$ where each u_i is contributed by party P_i. For simplicity of exposition, we restrict our discussion to the case where all the parties receive the same output.

Let $C : \{0,1\}^\lambda \times \ldots \times \{0,1\}^\lambda \to \{0,1\}^{\ell(\lambda)}$ be any n-input circuit that the parties are interested in computing. Then, for any input vector $\vec{x} = x_1, \ldots, x_n$, where $x_i \in \{0,1\}^\lambda$, the corresponding hardwired circuit is denoted as $C[\vec{x}]$. The corresponding hardwired circuit family is denoted as $\{\mathcal{C}[X]_\lambda\}_{\lambda \in \mathbb{N}}$ where $X = \{0,1\}^{n \times \lambda}$.

We now establish our notation for updates. For any set system of strings $\mathcal{U} = \{\mathcal{U}_\lambda\}_{\lambda \in \mathbb{N}}$, we say that $\mathcal{C}[X]$ is $(\mathsf{Upd}, \mathcal{U})$-updatable if $C'[\vec{x}'] \leftarrow \mathsf{Upd}\left(C[\vec{x}], \mathbf{u}\right)$, where $C[\vec{x}] \in \mathcal{C}[X]_\lambda, \mathbf{u} = (u_1, \ldots, u_n)$, $u_i \in \mathcal{U}_\lambda$, is such that $C'[\vec{x}']$ is also a hardwired circuit.

Finally, we specify our adversarial model for MPC. We consider polynomial-time adversaries who can statically corrupt up to $n-1$ parties. We consider security with abort.

Ideal World. We start by describing the ideal world for UMPC. Let $C \in \mathcal{C}$ be the initial circuit that the parties wish to compute.

Inputs: Each party P_i obtains an initial input x_i. The adversary $\mathcal{S}$ is given auxiliary input z. $\mathcal{S}$ selects a subset of the parties $M \subset \mathcal{P}$ to corrupt, and is given the inputs x_ℓ of each party $P_\ell \in M$.

Sending inputs to trusted party: Each honest party P_i sends its input x_i to the trusted party. For each corrupted party $P_i \in M$, the adversary may select any value x_i^* and send it to the ideal functionality.

Trusted party computes output: Let $x_1^*, ..., x_n^*$ be the inputs that were sent to the trusted party. Let $C[\vec{x^*}]_0$ be the hardwired circuit corresponding to the circuit C and input vector $\vec{x^*} = x_1^*, \ldots, x_n^*$. The trusted party sends the evaluation of $C[\vec{x^*}]_0$ to the adversary who replies with either `continue` or `abort`. If the adversary's message is `abort`, then the trusted party sends $\perp$ to all honest parties. Otherwise, it sends the evaluation of $C[\vec{x^*}]_0$ to all honest parties.

ℓth Update Phase: For every $\ell \in [q]$, where q is chosen by the adversary, the following is repeated sequentially:

- Each party P_i sends an update string $u_{i,\ell} \in \mathcal{U}_\lambda$ to the trusted party.
- The trusted party computes $C[\vec{x^*}]_\ell \leftarrow \mathsf{Upd}(C[\vec{x^*}]_{\ell-1}, \mathbf{u}_\ell)$ where $\mathbf{u}_\ell = (u_{1,\ell}, \ldots, u_{n,\ell})$. It sends the evaluation of $C[\vec{x^*}]_\ell$ to the adversary who replies with either `continue` or `abort`. If the adversary's message is `abort`, then the trusted party sends $\perp$ to all honest parties. Otherwise, it sends the evaluation of $C[\vec{x^*}]_\ell$ to all honest parties.

Outputs: Honest parties output all the messages they obtained from the ideal functionality. Malicious parties may output an arbitrary PPT function of the adversary's view.

The overall output of the ideal-world experiment consists of the outputs of all parties. For any ideal-world adversary $\mathcal{S}$ with auxiliary input $z \in \{0,1\}^*$, input vector $\vec{x}$, any arbitrary polynomial set of updates $\{\mathbf{u}_\ell\}_{\ell=1}^q$, and security parameter λ, we denote the output of the corresponding ideal-world experiment by

$$\mathsf{IDEAL}_{\mathcal{S},M}\Big(1^\lambda, \vec{x}, z, \{\mathbf{u}_\ell\}_{\ell=1}^q\Big).$$

Real World. The real world execution begins by an adversary $\mathcal{A}$ selecting any arbitrary subset of parties $M \subset \mathcal{P}$ to corrupt. The parties then engage in an execution of a real n-party updatable MPC protocol $\Pi = (\Pi_{\mathsf{init}}, \Pi_{\mathsf{upd}})$ for initial circuit $C \in \mathcal{C}$ that consists of two stages, namely, (a) an initial computation phase, (b) an *update phase*, where the latter can be repeated polynomially many times. Throughout the execution of Π, the adversary $\mathcal{A}$ sends all messages on behalf of the corrupted parties, and may follow an arbitrary polynomial-time strategy. In contrast, the honest parties follow the instructions of Π.

We now describe the two phases in the protocol.

Initial Computation Phase: Let x_i be the initial input of party P_i. In this phase, all the parties execute protocol Π_{init}, where each honest party $P_i \in \mathcal{P} \setminus M$ acts in accordance with its input x_i. At the end of the protocol, each honest party computes an initial output as well as state $\mathsf{st}_{i,0}$ in accordance with the protocol. If the protocol computation ends in an abort, then each honest party sets its output and state to $\perp$.

ℓth Update Phase: Whenever the parties wish to perform a computation over an updated circuit and input vector, they run an execution of Π_{upd}, where each honest party acts in accordance with its input $(\mathsf{st}_{i,\ell-1}, u_{i,\ell})$. At the end of the protocol, each honest party computes an output and an updated state st_ℓ, both of which may be set to $\perp$ if the protocol ends in an abort.

At the conclusion of all the update phases, each honest party P_i outputs all the outputs it obtained in the computations. Malicious parties may output an arbitrary PPT function of the view of $\mathcal{A}$.

For any adversary $\mathcal{A}$ with auxiliary input $z \in \{0,1\}^*$, input vector $\vec{x}$, any arbitrary polynomial set of updates $\{\mathbf{u}_\ell\}_{\ell=1}^q$, and security parameter λ, we denote the output of the multi-function MPC protocol $\Pi = (\Pi_{\mathsf{init}}, \Pi_{\mathsf{Upd}})$ by

$$\mathsf{REAL}^{\Pi}_{\mathcal{A},M}\Big(1^\lambda, \vec{x}, z, \{\mathbf{u}_\ell\}_{\ell=1}^q\Big).$$

Efficiency. We require the following efficiency property: the total communication complexity of any update phase ℓ is a fixed polynomial in the size of the update length $|\mathbf{u}_\ell|$ and the security parameter.

Security Definition. We say that a protocol Π is a UMPC protocol if any adversary, who corrupts a subset of parties and runs the protocol with honest parties, gains *no information* about the inputs of the honest parties beyond the protocol outputs that correspond to sequential evaluations of $C[\vec{x}], C[\vec{x}]_1, \ldots, C[\vec{x}]_q$ where $C[\vec{x}]_\ell = \mathsf{Upd}(C[\vec{x}], \mathbf{u}_\ell)$, $\mathbf{u}_\ell = (u_{1,\ell}, \ldots, u_{n,\ell})$.

Definition 28 (Updatable MPC)**.** *A protocol* $\Pi = (\Pi_{\mathsf{init}}, \Pi_{\mathsf{upd}})$ *is a secure n-party* UMPC *protocol for a* $(\mathsf{Upd}, \mathcal{U})$*-updatable circuit family* $\mathcal{C}$ *if for every PPT adversary* $\mathcal{A}$ *in the real world, there exists a PPT adversary* $\mathcal{S}$ *corrupting the same parties in the ideal world such that for every initial input vector* $\vec{x}$*, every auxiliary input* z*, and every sequence of updates* $\{\mathbf{u}_\ell\}_{\ell=1}^q$ *where* $\mathbf{u}_\ell = (u_{1,\ell}, \ldots, u_{n,\ell})$ *and* $u_{i,\ell} \in \mathcal{U}$*, it holds that*

$$\mathsf{IDEAL}_{\mathcal{S},M}\Big(1^\lambda, \vec{x}, z, \{\mathbf{u}_\ell\}_{\ell=1}^q\Big) \approx_c \mathsf{REAL}^{\Pi}_{\mathcal{A},M}\Big(1^\lambda, \vec{x}, z, \{\mathbf{u}_\ell\}_{\ell=1}^q\Big).$$

6.3.1 Construction of UMPC

In this section, we construct a UMPC protocol for general circuits.

Notation. Let $\mathcal{C}$ be a n-input $(\mathsf{Upd}, \mathcal{U})$-updatable circuit family and let $\mathcal{C}[X]$ denote the corresponding updatable hardwired circuit family. In order to construct a UMPC protocol for $\mathcal{C}$, we will use the following ingredients in our construction:

1. A stateless[12] URE scheme $(\mathsf{URE.Encode}, \mathsf{URE.GenUpd}, \mathsf{URE.ApplyUpd}, \mathsf{URE.Decode})$ for an $(\mathsf{Upd}_{\mathsf{ure}}, \mathcal{U}_{\mathsf{ure}})$-updatable class of hardwired circuits $\mathcal{C}[X]$ where $\mathcal{U}_{\mathsf{ure}} = \mathcal{U}^n$ and $\mathsf{Upd}_{\mathsf{ure}}$ is the same as Upd: for any $C[\vec{x}] \in \mathcal{C}[X]$ and any $\mathbf{u} \in \mathcal{U}_{\mathsf{ure}}$ where $\mathbf{u} = (u_1, \ldots, u_n)$, $\mathsf{Upd}_{\mathsf{ure}}(C[\vec{x}], \mathbf{u}) = \mathsf{Upd}(C[\vec{x}, \mathbf{u})$.
2. A standard n-party MPC protocol Π_{mpc} for general circuits that is secure against arbitrary, static corruptions.

[12]Our construction also works if we start with a stateful URE scheme. For simplicity of exposition, however, we present our construction using a stateless URE scheme.

Construction. We now describe our construction of a UMPC protocol $\Pi = (\Pi_{\mathsf{init}}, \Pi_{\mathsf{upd}})$.

Protocol Π_{init}: Let $C_0 \in \mathcal{C}$ be the level-0 circuit that the parties wish to compute and $x_{i,0}$ denote the level-0 input of party P_i. Protocol Π_{init} consists of the following two steps:

1. First, each party P_i privately samples a random string r_i.
2. Next, the parties engage in an execution of Π_{mpc} for computing the following function f_0: it takes as input $(C_0, x_{i,0}, r_i)$ from party P_i and computes $\langle C_0[\vec{x}_0]\rangle_{\mathsf{ure}} \leftarrow \mathsf{URE.Encode}(C_0[\vec{x}_0]; r)$ using randomness $r = r_1 \oplus \cdots \oplus r_n$, where $\vec{x}_0 = (x_{1,0}, \ldots, x_{n,0})$. The output of f is $\langle C_0[\vec{x}_0]\rangle_{\mathsf{ure}}$.
3. At the end of Π_{mpc}, each party computes $y_0 \leftarrow \mathsf{URE.Decode}(\langle C_0[\vec{x}_0]\rangle_{\mathsf{ure}})$ and outputs y_0. Each party P_i stores $\mathsf{st}_{i,0} = (\langle C[\vec{x}]\rangle_{\mathsf{ure}}, r_i)$.

Protocol Π_{upd}: Let $u_{i,\ell}$ denote the ℓth update string corresponding to party P_i. Let $\mathsf{st}_{i,\ell} = (\langle C_{\ell-1}[\vec{x_{\ell-1}}]\rangle_{\mathsf{ure}}, r_i)$ be the state of party P_i at the start of the ℓth update phase. Then, the ℓth execution of protocol Π_{upd} consists of the following two steps:

1. First, the parties engage in an execution of Π_{mpc} for computing the following (randomized) function f_ℓ: it takes as input $(u_{i,\ell}, r_i)$ from party P_i and computes $\langle \mathbf{u}_\ell\rangle_{\mathsf{ure}} \leftarrow \mathsf{URE.GenUpd}(\mathbf{u}_\ell, r)$ where $\mathbf{u}_\ell = (u_{1,\ell}, \ldots, u_{n,\ell})$ and $r = r_1 \oplus \cdots \oplus r_n$. The output of f_ℓ is $\langle \mathbf{u}_\ell\rangle_{\mathsf{ure}}$.
2. At the end of Π_{mpc}, each party computes $\langle C_\ell[\vec{x_\ell}]\rangle_{\mathsf{ure}} \leftarrow \mathsf{URE.ApplyUpd}(\langle C_{\ell-1}[\vec{x_{\ell-1}}]\rangle_{\mathsf{ure}}, \langle \mathbf{u}_\ell\rangle_{\mathsf{ure}})$.
3. Next, each party computes $y_\ell \leftarrow \mathsf{URE.Decode}(\langle C_\ell[\vec{x_\ell}]\rangle_{\mathsf{ure}})$ and outputs y_ℓ. Finally, P_i updates its state to $\mathsf{st}_i = (\langle C_\ell[\vec{x_\ell}]\rangle_{\mathsf{ure}}, r_i)$.

This completes the description of Π.

Proof Sketch. It is easy to see that the above construction satisfies our desired efficiency property. Specifically, since the ℓ^{th} update phase simply involves the execution of a standard MPC protocol π_{mpc} to compute an update encoding $\langle \mathbf{u}_\ell\rangle_{\mathsf{ure}}$, it follows from the efficiency property of URE that the the size of the function f_ℓ computed by the MPC protocol is a fixed polynomial in the size of $|\mathbf{u}_\ell|$ and the security parameter. It follows then from the efficiency of a standard MPC that the total communication complexity of the ℓ^{th} update phase is a fixed polynomial in the size of $|\mathbf{u}_\ell|$ and the security parameter.

Next, we argue security of our construction. We establish this by a simply hybrid argument. We start with hybrid H_0 that corresponds to the real world experiment. Next, we consider hybrid H_1 where we simulate all the executions in π_{mpc}. Note that here we still use the inputs and the updates of the honest parties. Finally, we consider hybrid H_2 where we simulate the output of each execution of π_{mpc}, i.e., we simulate the URE encoding $\langle C_0[\vec{x}_0]\rangle_{\mathsf{ure}}$ and the update encodings $\langle \mathbf{u}_\ell\rangle_{\mathsf{ure}}$ using the URE simulator who is provided the outputs using the trusted party in the ideal world. This experiment corresponds to the simulator's algorithm.

The indistinguishability of H_0 and H_1 follows easily from the security of the MPC protocol π_{mpc}. The indistinguishability of H_1 and H_2 follows from the security of URE. This completes our proof sketch.

References

[ABC+07] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 598–609. Acm, 2007.

[AFGH05] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.

[AGVW13] Shweta Agrawal, Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption: New perspectives and lower bounds. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 500–518, 2013.

[AIK07] Benny APPLEBAUM, Yuval ISHAI, and Eyal KUSHILEVITZ. Cryptogaphy in nc0. *SIAM journal on computing*, 36(4):845–888, 2007.

[AJ15] Prabhanjan Ananth and Abhishek Jain. Indistinguishability obfuscation from compact functional encryption. In *Advances in Cryptology–CRYPTO 2015*, pages 308–326. Springer, 2015.

[AJS15a] Prabhanjan Ananth, Abhishek Jain, and Amit Sahai. Indistinguishability obfuscation from functional encryption for simple functions. Technical report, Cryptology ePrint Archive, Report 2015/730, 2015.

[AJS15b] Prabhanjan Ananth, Abhishek Jain, and Amit Sahai. Patchable indistinguishability obfuscation: io for evolving software. *IACR Cryptology ePrint Archive*, 2015:1084, 2015.

[BBS98] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In *Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques, Espoo, Finland, May 31 - June 4, 1998, Proceeding*, pages 127–144, 1998.

[BF11] Dan Boneh and David Mandell Freeman. Homomorphic signatures for polynomial functions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 149–168. Springer, 2011.

[BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In *Advances in Cryptology—CRYPTO'94*, pages 216–233. Springer, 1994.

[BGG95] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography and application to virus protection. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 45–56. ACM, 1995.

[BGI$^+$01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2001.

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *Public-Key Cryptography–PKC 2014*, pages 501–519. Springer, 2014.

[BGL$^+$15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Siddartha Telang. Succinct randomized encodings and their applications. In *STOC*, 2015.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796. ACM, 2012.

[BKY01] Enrico Buonanno, Jonathan Katz, and Moti Yung. Incremental unforgeable encryption. In *Fast Software Encryption*, pages 109–124. Springer, 2001.

[BLMR13] Dan Boneh, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. Key homomorphic prfs and their applications. In *Advances in Cryptology–CRYPTO 2013*, pages 410–428. Springer, 2013.

[BM97] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Advances in Cryptology—EUROCRYPT'97*, pages 163–192. Springer, 1997.

[BM99] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 431–448, 1999.

[BNPW16] Nir Bitansky, Ryo Nishimaki, Alain Passelègue, and Daniel Wichs. From cryptomania to obfustopia through secret-key functional encryption. volume TCC, 2016.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *Theory of Cryptography*, pages 253–273. Springer, 2011.

[BV15a] Nir Bitansky and Vinod Vaikuntanathan. Indistinguishability obfuscation from functional encryption. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 171–190. IEEE, 2015.

[BV15b] Zvika Brakerski and Vinod Vaikuntanathan. Constrained key-homomorphic prfs from standard lattice assumptions. In *Theory of Cryptography*, pages 1–30. Springer, 2015.

[BW07] Dan Boneh and Brent Waters. Conjunctive, subset, and range queries on encrypted data. In *Theory of Cryptography Conference*, pages 535–554. Springer, 2007.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *Advances in Cryptology-ASIACRYPT 2013*, pages 280–300. Springer, 2013.

[CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Indistinguishability obfuscation of iterated circuits and RAM programs. In *STOC*, 2015.

[CIJ$^+$13] 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 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 519–535, 2013.

[CKLM12] Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Sarah Meiklejohn. Malleable proof systems and applications. In *Advances in Cryptology–EUROCRYPT 2012*, pages 281–300. Springer, 2012.

[CKLM13] Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Sarah Meiklejohn. Succinct malleable nizks and an application to compact shuffles. In *Theory of Cryptography*, pages 100–119. Springer, 2013.

[CKO14] Nishanth Chandran, Bhavana Kanukurthi, and Rafail Ostrovsky. Locally updatable and locally decodable codes. In *Theory of Cryptography*, pages 489–514. Springer, 2014.

[DSLSZ15] Dana Dachman-Soled, Feng-Hao Liu, Elaine Shi, and Hong-Sheng Zhou. Locally decodable and updatable non-malleable codes and their applications. In *Theory of Cryptography*, pages 427–450. Springer, 2015.

[DvOW92] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Des. Codes Cryptography*, 2(2):107–125, 1992.

[Fis97] Marc Fischlin. Incremental cryptography and memory checkers. In *Advances in Cryptology—EUROCRYPT'97*, pages 393–408. Springer, 1997.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178, 2009.

[GGG+14] Shafi Goldwasser, S. Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 578–602. Springer, 2014.

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 40–49. IEEE Computer Society, 2013.

[GGHZ14] Sanjam Garg, Craig Gentry, Shai Halevi, and Mark Zhandry. Fully secure functional encryption without obfuscation. *IACR Cryptology ePrint Archive*, 2014:666, 2014.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 1986.

[GGSW13] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 467–476. ACM, 2013.

[GHL+14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled ram revisited. In *Advances in Cryptology–EUROCRYPT 2014*, pages 405–422. Springer, 2014.

[GHV10] Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. i-hop homomorphic encryption and rerandomizable yao circuits. In *Advances in Cryptology–CRYPTO 2010*, pages 155–172. Springer, 2010.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 555–564. ACM, 2013.

[GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled ram. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 210–229. IEEE, 2015.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled ram from one-way functions. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 449–458. ACM, 2015.

[GLSW14] Craig Gentry, Allison B. Lewko, Amit Sahai, and Brent Waters. Indistinguishability obfuscation from the multilinear subgroup elimination assumption. *IACR Cryptology ePrint Archive*, 2014:309, 2014.

[GLW14] Craig Gentry, Allison Lewko, and Brent Waters. Witness encryption from instance independent assumptions. In *International Cryptology Conference*, pages 426–443. Springer, 2014.

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

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 182–194, 1987.

[GP15] Sanjam Garg and Omkant Pandey. Incremental program obfuscation. *IACR Cryptology ePrint Archive*, 2015:997, 2015.

[GPSW06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, Ioctober 30 - November 3, 2006*, pages 89–98, 2006.

[Gün89] Christoph G. Günther. An identity-based key-exchange protocol. In *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings*, pages 29–37, 1989.

[GVW12] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption with bounded collusions via multi-party computation. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 162–179, 2012.

[GVW15a] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Predicate encryption for circuits from lwe. In *Annual Cryptology Conference*, pages 503–523. Springer, 2015.

[GVW15b] Sergey Gorbunov, Vinod Vaikuntanathan, and Daniel Wichs. Leveled fully homomorphic signatures from standard lattices. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, pages 469–477. ACM, 2015.

[HJO$^+$15] Brett Hemenway, Zahra Jafargholi, Rafail Ostrovsky, Alessandra Scafuro, and Daniel Wichs. Adaptively secure garbled circuits from one-way functions. *IACR Cryptology ePrint Archive*, 2015:1250, 2015.

[HLP11] Shai Halevi, Yehuda Lindell, and Benny Pinkas. Secure computation on the web: Computing without simultaneous interaction. In *Advances in Cryptology–CRYPTO 2011*, pages 132–150. Springer, 2011.

[IK00] Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 294–304. IEEE, 2000.

[KPTZ13] 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.

[KSW08] Jonathan Katz, Amit Sahai, and Brent Waters. Predicate encryption supporting disjunctions, polynomial equations, and inner products. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 146–162. Springer, 2008.

[LO13] Steve Lu and Rafail Ostrovsky. How to garble ram programs? In *Advances in Cryptology–EUROCRYPT 2013*, pages 719–734. Springer, 2013.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[LPST16a] Huijia Lin, Rafael Pass, Karn Seth, and Sidharth Telang. Indistinguishability obfuscation with non-trivial efficiency. In *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part II*, pages 447–462, 2016.

[LPST16b] Huijia Lin, Rafael Pass, Karn Seth, and Sidharth Telang. Output-compressing randomized encodings and applications. In *Theory of Cryptography*, pages 96–124. Springer, 2016.

[Mic97] Daniele Micciancio. Oblivious data structures: applications to cryptography. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 456–464. ACM, 1997.

[MPRS12] Ilya Mironov, Omkant Pandey, Omer Reingold, and Gil Segev. Incremental deterministic public-key encryption. In *Advances in Cryptology–EUROCRYPT 2012*, pages 628–644. Springer, 2012.

[O'N10] Adam O'Neill. Definitional issues in functional encryption. *IACR Cryptology ePrint Archive*, 2010:556, 2010.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious rams. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 514–523, 1990.

[SBC$^+$07] Elaine Shi, John Bethencourt, TH Hubert Chan, Dawn Song, and Adrian Perrig. Multi-dimensional range query over encrypted data. In *2007 IEEE Symposium on Security and Privacy (SP'07)*, pages 350–364. IEEE, 2007.

[SS10] Amit Sahai and Hakan Seyalioglu. Worry-free encryption: functional encryption with public keys. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 463–472. ACM, 2010.

[SW05] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, pages 457–473, 2005.

[SW14] 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.

[Wat15] Brent Waters. A punctured programming approach to adaptively secure functional encryption. In *CRYPTO 2015*, 2015.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

[Zha16] Mark Zhandry. How to avoid obfuscation using witness prfs. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 421–448, 2016.

A Other Definitions of URE

We can consider updating processes different from the sequential process that we studied in Section 3.1. Another important type of updating process is parallel updating that we define next.

A.1 Parallel Updating

Given a randomized encoding $\langle C[x] \rangle_{\mathsf{ure}}$ and multiple patches, parallel updating allows for separately updating the original encoding using each of these patches. That is, given secure updates $\langle \mathbf{u}_1 \rangle_{\mathsf{ure}}, \ldots, \langle \mathbf{u}_p \rangle_{\mathsf{ure}}$, we can update $\langle C[x] \rangle_{\mathsf{ure}}$ to obtain the respective updated encodings $\langle C_1[x_1] \rangle_{\mathsf{ure}}, \ldots, \langle C_p[x_p] \rangle_{\mathsf{ure}}$, where for each i, $\langle C_i[x_i] \rangle_{\mathsf{ure}} = \mathsf{ApplyUpd}(\langle C[x] \rangle_{\mathsf{ure}}, \langle \mathbf{u}_i \rangle_{\mathsf{ure}})$. We emphasize that this process does not immediately allow for updating the already updated randomized encoding $\langle C_i[x_i] \rangle_{\mathsf{ure}}$ again. This is in contrast with the sequential updating process where an updated randomized encoding can indeed be updated again.

Correctness of Parallel Updating. Intuitively, the correctness property states that whether you first compute the randomized encoding $\langle C[x] \rangle_{\mathsf{ure}}$ and then apply the secure update $\langle \mathbf{u} \rangle_{\mathsf{ure}}$ or if you first update the hardwired circuit $C[x]$ using $\mathbf{u}$ and then compute the randomized encoding, the decoding of either one of them yields the same output. In the formal description below we take into account multiple parallel updates.

Consider a circuit $C \in \mathcal{C}_\lambda$, input $x \in \{0,1\}^\lambda$. Let $\mathbf{U} \in (\mathcal{U}_\lambda)^p$ be a vector consisting of update strings, where $p(\lambda)$ is polynomial in λ. Consider the following two processes:

Secure updating process: This corresponds to the case when you compute the randomized encoding first and then apply the secure updates.

1. $(\langle C[x] \rangle_{\mathsf{ure}}, \mathsf{st}) \leftarrow \mathsf{Encode}\left(1^\lambda, C, x\right)$.
2. For every $i \in [p]$; $(\langle \mathbf{u}_i \rangle_{\mathsf{ure}}, \mathsf{st}') \leftarrow \mathsf{GenUpd}\left(\mathsf{st}, \mathbf{u}_i\right)$, where $\mathbf{u}_i$ is the i^{th} entry in $\mathbf{U}$.
3. For every $i \in [p]$; $\langle C_i[x_i] \rangle_{\mathsf{ure}} \leftarrow \mathsf{ApplyUpd}\left(\langle C[x] \rangle_{\mathsf{ure}}, \langle \mathbf{u}_i \rangle_{\mathsf{ure}}\right)$.

Insecure updating process: This corresponds to the case when you first update the circuit and then compute a fresh randomized encoding.

1. For every $i \in [p]$, we have $C_i[x_i] \leftarrow \mathsf{Upd}(C[x], \mathbf{u}_i)$. We have $C_p(x_p)$ to be the output of $C_p[x_p]$.

We have,

$$\mathsf{Decode}\left(\langle C_p[x_p] \rangle_{\mathsf{ure}}\right) = C_p(x_p)$$

Security of Parallel Updating. There are different ways to formalize the security of updatable randomized encodings. Inspired by the literature on randomized encodings, we study two security notions - namely, simulation-based and indistinguishability-based.

Simulation-Based Security. We adopt the *real world/ ideal world* paradigm in formalizing the simulation-based security definition of parallel updatable RE. In the real world, the adversary receives a randomized encoding and encodings of updates. All the encodings are generated honestly as per the description of the scheme. In the ideal world, the adversary is provided simulated randomized encodings and encodings of updates. These simulated encodings are generated as a function of the outputs and in particular, the simulation process does not receive as input the circuit, input or the plaintext updates. A parallel updatable RE scheme is secure if an efficient adversary cannot tell apart real world from the ideal world.

The ideal world is formalized by considering a simulator Sim that runs in probabilistic polynomial time. Sim gets as input the output of circuit $C(x)$, the length of C and produces a simulated randomized encoding. We emphasize that Sim does not receive as input C or x. After this, Sim simulates the update encodings. On input length of update $\mathbf{u}_i$, value $C_i(x_i)$, it generates a simulated encoding of $\mathbf{u}_i$. Here, $C_i(x_i)$ is obtained by first updating $C[x]$ using $\mathbf{u}_i$ to obtain $C_i[x_i]$, whose output is $C_i(x_i)$. For this discussion, we consider the scenario where the circuit, input along with the updates are fixed at the beginning of the experiment. This is termed as the *selective* setting. We describe the formal experiment in Figure 6.

We present the formal security definition below.

Definition 29 (SIM-secure Parallel URE)**.** *A parallel URE scheme* URE *for* $(\mathsf{Upd},\mathcal{U})$*-updatable class of circuits* $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda\in\mathbb{N}}$ *is said to be* **SIM-secure** *if for every PPT adversary* $\mathcal{A}$*, for every circuit* $C \in \mathcal{C}_\lambda$*, updates* $\mathbf{u}_1, \ldots, \mathbf{u}_p \in \mathcal{U}_\lambda$*, there exists a PPT simulator* Sim *such that the following holds for sufficiently large* $\lambda \in \mathbb{N}$*,*

$$\left|\mathsf{Pr}\left[0 \leftarrow \mathsf{IdealExpt}^{\mathcal{A}}\left(1^\lambda, C, x, \{\mathbf{u}_i\}_{i\in[p]}\right)\right] - \mathsf{Pr}\left[0 \leftarrow \mathsf{RealExpt}^{\mathcal{A}}\left(1^\lambda, C, x, \{\mathbf{u}_i\}_{i\in[p]}\right)\right]\right| \leq \mathsf{negl}(\lambda),$$

where negl *is a negligible function.*

<u>**Experiment** $\mathsf{IdealExpt}^{\mathcal{A}}(1^\lambda, C, x, \{\mathbf{u}_i\}_{i\in[p]})$:</u>	<u>**Experiment** $\mathsf{RealExpt}(1^\lambda, C, x, \{\mathbf{u}_i\}_{i\in[p]})$:</u>
1. $(\langle C[x]\rangle_{\mathsf{ure}}, \mathsf{st}) \leftarrow \mathsf{Sim}(1^\lambda, 1^{\lvert C\rvert}, C(x))$.	1. $(\langle C[x]\rangle_{\mathsf{ure}}, \mathsf{st}) \leftarrow \mathsf{Encode}\left(1^\lambda, C, x\right)$.
2. $C[x] :=$ hardwired circuit of (C, x).	2. $\forall i \in [p]$, $(\langle \mathbf{u}_i\rangle_{\mathsf{ure}}, \mathsf{st}_i) \leftarrow \mathsf{GenUpd}\left(\mathsf{st}, \mathbf{u}_i\right)$.
3. $\forall i \in [p]$, $C_i[x_i] \leftarrow \mathsf{Upd}(C[x], \mathbf{u}_i)$. Let $C_i(x_i)$ be the output computed by $C_i[x_i]$.	
4. $\forall i \in [p]$, $\langle \mathbf{u}_i\rangle_{\mathsf{ure}} \leftarrow \mathsf{Sim}(\mathsf{st}, 1^{\lvert\mathbf{u}_i\rvert}, C_i[x_i])$.	
Output $\mathcal{A}\Big(\langle C[x]\rangle_{\mathsf{ure}}, \langle \mathbf{u}_1\rangle_{\mathsf{ure}}, \ldots, \langle \mathbf{u}_p\rangle_{\mathsf{ure}}\Big)$.	Output $\mathcal{A}\Big(\langle C[x]\rangle_{\mathsf{ure}}, \langle \mathbf{u}_1\rangle_{\mathsf{ure}}, \ldots, \langle \mathbf{u}_p\rangle_{\mathsf{ure}}\Big)$.

Figure 6: Selective Simulation-Based Definition of Parallel Updatable RE.

Indistinguishability-based Security. We formulate a game-based definition between the challenger and the adversary. The adversary makes circuit query (C^0, C^1) along with input x to the challenger. The challenger picks a bit b at random and encodes (C^b, x). This challenge encoding is sent to the adversary. The adversary also queries for secure updates. That is, it sends the pair $(\mathbf{u}_i^0, \mathbf{u}_i^1)$ to the challenger who responds with encoding of $\mathbf{u}_i^b$. The adversary is restricted to "valid" update queries: it should hold that $C_i^0[x_i^0] = C_i^1[x_i^1]$ for every i, where $C_i^0[x_i^0] \leftarrow \mathsf{Upd}(C^0[x], \mathbf{u}_i^0)$ and $C_i^1[x_i^1] \leftarrow \mathsf{Upd}(C^1[x], \mathbf{u}_i^1)$. If the adversary makes any invalid update queries, the challenger aborts the experiment. In the end, the adversary is required to guess the bit b. We say that the parallel URE scheme is IND-secure if the adversary succeeds with negligible advantage (i.e., with probability negligibly close to 1/2).

We can consider different flavors of IND-based security depending on the order of circuit and update queries made by the adversary. We consider the simplest setting where the adversary is supposed to declare all his queries in the beginning of the game. We call this *selective* setting. We can also consider the *adaptive* setting where the adversary can chose the circuit and the update queries adaptively.

<u>$\mathsf{Expt}_{\mathcal{A}}^{\mathsf{PURE}}(1^\lambda, b)$:</u>

- $\mathcal{A}$ sends circuits $(C^0, C^1) \in \mathcal{C}_\lambda$, input $x \in \{0,1\}^\lambda$ to Ch.
- $\mathcal{A}$ additionally sends the update queries $(\mathbf{u}_i^0, \mathbf{u}_i^1)$ for $i \in [p(\lambda)]$ to Ch. Challenger first checks if the following condition holds: letting $C_i^\beta[x_i] \leftarrow \mathsf{Upd}(C^\beta[x], \mathbf{u}_i^\beta)$ for $\beta \in \{0,1\}$ and

$i \in [p]$:

$$\Big(C^0(x) = C^1(x)\Big) \text{ and } \Big(\forall i \in [p]:\ C_i^0(x_i^0) = C_i^1(x_i^0)\Big).$$

Here, $C_i^0(x_i)$ (resp., $C_i^1(x_i)$) denotes the output of $C_i^0[x_i]$ (resp., $C_i^1[x_i]$). If the above condition is not satisfied, Ch aborts the experiment.

- Ch sends $\langle C^b[x]\rangle_{\mathsf{ure}}$ to $\mathcal{A}$, where $(\langle C^b[x]\rangle_{\mathsf{ure}}, \mathsf{st}) \leftarrow \mathsf{Encode}(1^\lambda, C^b, x)$, to $\mathcal{A}$. Also, Ch sends $(\langle \mathbf{u}_i^b\rangle_{\mathsf{ure}}, \mathsf{st}') \leftarrow \mathsf{GenUpd}(\mathsf{st}, \mathbf{u}_i^b)$ for every $i \in [p(\lambda)]$.
- $\mathcal{A}$ outputs b'.

We give the formal definition of the security notion below.

Definition 30 (IND-secure Parallel URE)**.** *A parallel* URE *scheme is* **IND-secure** *if for any PPT adversary $\mathcal{A}$, bit b, we have* $\mathsf{Pr}[b' = b : b' \leftarrow \mathsf{Expt}_{\mathcal{A}}^{\mathsf{PURE}}(1^\lambda, b)] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$*, for some negligible function* negl.

A.2 Connection between Parallel URE and Reusable Garbled Circuits

The notions of parallel URE and reusable garbled circuits are closely connected. Specifically, consider an *input-updatable* family of hardwired circuits (defined formally below), in which an update is simply an input x', and $\mathsf{Upd}(C[x], x') = C[x']$. By a simple transformation, the existence of a parallel URE scheme for the hardwired circuit family $\{\mathcal{C}[X]_\lambda\}_{\lambda \in \mathbb{N}}$ is equivalent to the existence of a reusable garbled circuit scheme for $\mathcal{C}$. We present the transformation from a parallel URE scheme to a reusable garbling scheme; the reverse transformation is as straightforward.

Let $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda \in \mathbb{N}}$ be a family of circuits, and $\{\mathcal{C}[X]_\lambda\}_{\lambda \in \mathbb{N}}$ be the corresponding family of hardwired circuits (Definition 8). We $\{\mathcal{C}[X]_\lambda\}_{\lambda \in \mathbb{N}}$ is *input-updatable* if it is $(\mathcal{U}_{\mathsf{inp}}, \mathsf{Upd}_{\mathsf{inp}})$-updatable:

- $\mathcal{U}_{\mathsf{inp}} = \{0,1\}^*$.
- $\mathsf{Upd}_{\mathsf{inp}}(C[x], x')$ takes as input a hardwired circuit $C[x] \in \mathcal{C}[X]$ and a update $x' \in \mathcal{U}_{\mathsf{inp}}$ indicating the new input to hardwired. If $|x| > \lambda$, then $\mathsf{Upd}_{\mathsf{inp}}$ outputs $\perp$. Otherwise, output the hardwired circuit $C[x']$.

Observe that in the sequential URE setting, the ability to compose updates implies that to update the input of a hardwired circuit, it suffices to support only bitwise updates, in which only a single bit is changed. In the setting of parallel updating, this is no longer the case.

Suppose $\mathsf{URE} = (\mathsf{Encode}, \mathsf{GenUpd}, \mathsf{ApplyUpd}, \mathsf{Decode})$ is a secure parallel URE scheme for the input-updatable class of hardwired circuits $\{\mathcal{C}[X]_\lambda\}_{\lambda \in \mathbb{N}}$. We construct a reusable garbled circuit scheme for $\mathcal{C}$ as follows:

- $\mathsf{GrbCkt}(1^\lambda, C) = \mathsf{Encode}(1^\lambda, C, 0^\lambda)$. This outputs a state $\mathsf{st}_{\mathsf{gc}} = \mathsf{st}_{\mathsf{ure}}$ and a garbled circuit $\langle C\rangle_{\mathsf{gc}} = \langle C[0^\lambda]\rangle_{\mathsf{ure}}$.
- $\mathsf{GrbInp}(\mathsf{st}, x) = \mathsf{GenUpd}(\mathsf{st}, x)$. This outputs a garbled input $\langle x\rangle_{\mathsf{gc}} = \langle x\rangle_{\mathsf{ure}}$.
- $\mathsf{EvalGC}(\langle C\rangle_{\mathsf{gc}}, \langle x\rangle_{\mathsf{gc}}) = \mathsf{Decode}\left(\mathsf{ApplyUpd}(\langle C[0^\lambda]\rangle_{\mathsf{ure}}, \langle x\rangle_{\mathsf{ure}})\right)$.

The security of the URE scheme transfers directly to the security of the resulting reusable garbling scheme.

B Achieving Update Hiding Generically

We describe how to achieve update hiding generically. We give this transformation for the general case of updatable randomized encodings but the same transformation works even for updatable garbled circuits. The main idea is to use a non-interactive write-only oblivious RAM scheme to achieve this. We define this notion below.

Non-Interactive Write-Only ORAM. A scheme $\mathsf{wNIORAM}$ consists of the following algorithms.

- **Database Encoding,** $\mathsf{EncDB}(1^\lambda, D)$: On input security parameter λ, database D, it produces an encoded $\widetilde{D}$ and secret key osk. The database encoding is given to the server by the client. The client hides osk from the server.
- **Query encoding,** $\mathsf{EncQ}(\mathsf{osk}, q)$: On input secret key osk and write query q, it produces program encoding $\widetilde{q}$. The client sends the query encoding to the server. Here, we only consider write queries of the form $(index, b)$.
- **Updating,** $\mathsf{Upd}(\widetilde{q}, \widetilde{D})$: On input query encoding $\widetilde{q}$ and database encoding $\widetilde{D}$, it produces an updated database encoding $\widetilde{D'}$. Note that the server can execute this procedure non-interactively and in particular, this does not involve any communication with the client.

The correctness property is stated as in an oblivious RAM scheme – updating an encoding of the database using an encoded query is equivalent to updating the underlying database and then encoding it. The security guarantee (as in any ORAM scheme) is that the access pattern does not leak any information about the underlying query.

In this work, we are interested in $\mathsf{wNIORAM}$ schemes satisfying two properties:

1. The query encoding $\widetilde{q}$ is a tuple of pairs of the form $(index, b)$ and the updating algorithm substitutes b in $index^{th}$ position of $\widetilde{D}$ for every $(index, b)$ in $\widetilde{q}$. We call such schemes *bitwise* $\mathsf{wNIORAM}$ schemes.
2. Another property we are interested is the *decodability* property: given a valid database $\widetilde{D}$ and secret key osk, there is a public algorithm to recover D correctly.

The garbled RAM scheme of [GLOS15] yields a bitwise $\mathsf{wNIORAM}$ schemes with decodability property. Furthermore, their scheme is based on one-way functions. We thus have the following theorem.

Theorem 10. *There exists bitwise* $\mathsf{wNIORAM}$ *schemes with decodability property assuming one-way functions.*

Update-Hiding Transformation. Suppose we have an updatable randomized encoding scheme $\mathsf{URE}_{\mathsf{nph}}$ which does not necessarily satisfy update hiding property. We show how to achieve updatable RE, denoted by URE, with update hiding property.

$\mathsf{Encode}\left(1^\lambda, C, x\right)$: First, it computes $(\widetilde{(C,x)}, \mathsf{osk}) \leftarrow \mathsf{EncDB}(1^\lambda, (C,x))$. It executes $(\mathsf{URE}_{\mathsf{nph}}.\langle C^*[\perp]\rangle_{\mathsf{ure}}, \mathsf{st}) \leftarrow \mathsf{URE}_{\mathsf{nph}}.\mathsf{Encode}(1^\lambda, C^*, \perp)$, where C^* (with $\widetilde{(C,x)}$ and osk hardwired into it) is defined as follows. C^* first decodes $\widetilde{(C,x)}$ using osk to recover (C,x) (using decodability property) and then it outputs $C(x)$.

Finally, it outputs $\mathsf{URE}_{\mathsf{nph}}.\langle C^*[\perp]\rangle_{\mathsf{ure}}$ as the randomized encoding and it sets the state $\mathsf{st} = \mathsf{osk}$.

$\mathsf{GenUpd}\left(\mathsf{st}, \mathbf{u}\right)$: It first encodes $\mathbf{u}$ using the query encode algorithm EncQ to obtain $\widetilde{\mathbf{u}}$. It then compiles $\widetilde{\mathbf{u}}$ into an encoding $\mathsf{URE}_{\mathsf{nph}}.\langle\widetilde{\mathbf{u}}\rangle_{\mathsf{ure}}$ by executing $\langle\widetilde{\mathbf{u}}\rangle_{\mathsf{ure}} \leftarrow \mathsf{URE}_{\mathsf{nph}}.\mathsf{GenUpd}(\mathsf{st}, \widetilde{\mathbf{u}})$. The encoding of update is set to be $\mathsf{URE}_{\mathsf{nph}}.\langle\widetilde{\mathbf{u}}\rangle_{\mathsf{ure}}$. The new state is the same as the old state.

$\mathsf{ApplyUpd}\left(\langle C^*[x]\rangle_{\mathsf{ure}}, \langle\widetilde{\mathbf{u}}\rangle_{\mathsf{ure}}\right)$: This procedure essentially executes $\mathsf{ApplyUpd}$ of the scheme $\mathsf{URE}_{\mathsf{nph}}$. That is, it executes $\mathsf{URE}_{\mathsf{nph}}.\mathsf{ApplyUpd}(\mathsf{URE}_{\mathsf{nph}}.\langle C^*[\perp]\rangle_{\mathsf{ure}}, \mathsf{URE}_{\mathsf{nph}}.\langle\widetilde{\mathbf{u}}\rangle_{\mathsf{ure}})$ to obtain $\mathsf{URE}_{\mathsf{nph}}.\left\langle C^{*\prime}[\perp]\right\rangle_{\mathsf{ure}}$. It outputs the updated randomized encoding $\mathsf{URE}_{\mathsf{nph}}.\left\langle C^{*\prime}[\perp]\right\rangle_{\mathsf{ure}}$. Here, $C^{*\prime}$ contains the updated ORAM and similar to circuit C^* – it first decrypts the ORAM to get (C', x') and performs the computation $C'(x')$.

$\mathsf{Decode}\,(\langle C[x]\rangle_{\mathsf{ure}})$: This procedure executes the decoding procedure of $\mathsf{URE}_{\mathsf{nph}}$.

The correctness property of $\mathsf{URE}_{\mathsf{nph}}$ as well as the decodibility property of $\mathsf{wNIORAM}$ implies the correctness property of the above URE scheme. Furthermore, we can invoke the security property of $\mathsf{URE}_{\mathsf{nph}}$ and the access pattern hiding property of $\mathsf{wNIORAM}$ to show the security of the above scheme.

C Reducing the State of Authority

There is a generic approach to reduce the state size of the authority (i.e., the encoder in the definitions of URE and UGC) using garbled RAMs with persistent memory [GHL$^+$14, GLOS15]. Interestingly, our approach works even if the garbled RAM scheme is *non-succinct*, i.e., if the size of the program encoding is dependent on the computation time. Since the existence of such garbled RAM schemes can be based on one-way functions, we only need to assume the existence of one-way functions for our transformation.

We remark that the idea of using garbled RAMs to reduce the state size of the encoder was observed in [AJS15b]. However, their work crucially use a strong notion of succinct garbled RAMs whose existence is known only from indistinguishability obfuscation. In contrast, here, we only rely on non-succinct garbled RAM.

We now state our theorem:

Theorem 11. *Assuming that one-way functions exist, there exists an efficient transformation that transforms any URE (resp., UGC) scheme* URE *satisfying efficient update generation time property into a new URE (resp., UGC) scheme that also satisfies the state-size property.*

The main idea behind proving the above theorem is that the authority delegates the job of computing the secure update to the client. This delegation process is carried out by initially garbling the circuit and input pair using the garbled database encoding algorithm. This will be shipped as part of the initial randomized encoding. That is the randomized encoding of (C, x) w.r.t URE^* comprises of the garbled database encoding of (C, x) and randomized encoding of (C, x) w.r.t URE. During the update phase, the update generation algorithm of URE^* essentially encodes the update generation algorithm of URE using the garbled program encoding algorithm. On the client's end, the evaluation of the program encoding is done on the database to obtain the secure update computed w.r.t the old scheme URE. Using this secure update, the randomized encoding of (C, x) w.r.t URE is updated. The database encoding is also correspondingly updated (this is done as part of garbled evaluation algorithm) to correspond to the updated circuit and input pair.

We refer the reader to [AJS15b] for further details.