Output Compression, MPC, and iO for Turing Machines

Saikrishna Badrinarayanan [*] Rex Fernando [†] Venkata Koppula [‡] Amit Sahai [§]
Brent Waters [¶]

Abstract

In this work, we study the fascinating notion of output-compressing randomized encodings for Turing Machines, in a *shared randomness model*. In this model, the encoder and decoder have access to a shared random string, and the efficiency requirement is, the size of the encoding must be independent of the running time and output length of the Turing Machine on the given input, while the length of the shared random string is allowed to grow with the length of the output. We show how to construct output-compressing randomized encodings for Turing machines in the shared randomness model, assuming iO for circuits and any assumption in the set {LWE, DDH, Nth Residuosity}.

We then show interesting implications of the above result to basic feasibility questions in the areas of secure multiparty computation (MPC) and indistinguishability obfuscation (iO):

1. **Compact MPC for Turing Machines in the Random Oracle Model.** In the context of MPC, we consider the following basic feasibility question: does there exist a malicious-secure MPC protocol for Turing Machines whose communication complexity is independent of the running time and output length of the Turing Machine when executed on the combined inputs of all parties? We call such a protocol as a *compact* MPC protocol. Hubácek and Wichs [HW15] showed via an incompressibility argument, that, even for the restricted setting of circuits, it is impossible to construct a malicious secure two party computation protocol in the plain model where the communication complexity is independent of the output length. In this work, we show how to evade this impossibility by compiling any (non-compact) MPC protocol in the plain model to a *compact* MPC protocol for Turing Machines in the Random Oracle Model, assuming output-compressing randomized encodings in the shared randomness model.
2. **Succinct iO for Turing Machines in the Shared Randomness Model.** In all existing constructions of iO for Turing Machines, the size of the obfuscated program grows with a bound on the input length. In this work, we show how to construct an iO scheme for Turing Machines in the shared randomness model where the size of the obfuscated program is independent of a bound on the input length, assuming iO for circuits and any assumption in the set {LWE, DDH, Nth Residuosity}.

*UCLA. Email: saikrishna@cs.ucla.edu
†UCLA. Email: rex@cs.ucla.edu
‡UT Austin. Email: kvenkata@cs.utexas.edu.
§UCLA. Email: sahai@cs.ucla.edu.
¶UT Austin. Email: bwaters@cs.utexas.edu.

Contents

1 Introduction

In this work, we study the fascinating notion of output-compressing randomized encodings for Turing machines. We explore the implication of such encodings to a natural and surprisingly unexplored form of secure multiparty computation for Turing Machines, and also to indistinguishability obfuscation for Turing Machines.

Output-compressing randomized encodings were introduced in the works of Ananth and Jain[AJ15] and Lin, Pass, Seth and Telang [LPST16] as a generalization of randomized encodings [IK00] and succinct randomized encodings [KLW15, BGL$^+$15, CHJV15]. Recall that in an output-compressing randomized encoding scheme for Turing machines, there exists an encode algorithm that takes as input a Turing machine M and an input x. It outputs an encoding $\widetilde{M_x}$ such that the decode algorithm, given this encoding $\widetilde{M_x}$, can compute the output $M(x)$. The efficiency requirement is that for any machine M and input x, the size of the encoding is $\mathsf{poly}(|M|, |x|, \lambda)$, for some fixed polynomial poly, where λ is the security parameter. In particular, the size of the encoding should be *independent of the output length* and the running time of the machine M on input x.[1] In those papers, they defined both indistinguishability based and simulation based security notions. In this work, we will focus on the stronger notion of simulation based security which states that an output-compressing randomized encoding scheme is secure if there exists a simulator Sim, that, for any Turing machine M and input x, given just the output $M(x)$, along with the size of the machine $|M|$ and the input length $|x|$, outputs a simulated encoding $\widetilde{M_x}$ that is indistinguishable from a real encoding of the machine M and input x.[2] As stated here, this goal is impossible due to an "incompressibility" argument as shown by Lin et al.[LPST16]. Such incompressibility arguments have been a source of impossibility proofs in many areas of cryptography such as functional encryption, garbled circuits and secure multiparty computation [BSW11, AIKW13, CIJ$^+$13, AGVW13, HW15] and this is perhaps the reason why simulation secure output compressing randomized encodings have not been well studied so far.

Our starting observation is that the above impossibility fails to hold in a *shared randomness model* where the size of the randomness can grow with the output length. More formally, both the encoder and decoder share a random string (whose size can grow with the output length) and we require two properties: (1) For any machine M and input x, the size of the encoding is $\mathsf{poly}(|M|, |x|, \lambda)$, for some fixed polynomial poly. (2) There exists a simulator Sim, that, for any Turing machine M and input x, given just the output $M(x)$, along with the length of the machine $|M|$ and the input length $|x|$, outputs a pair of a simulated encoding $\widetilde{M_x}$ and a shared random string that is indistinguishable from the pair of a real encoding and a uniformly random string.

In fact, our first main result is that we can indeed construct output-compressing randomized encodings for Turing machines in the shared randomness model based on indistinguishability obfuscation (iO) for circuits and any assumption in {Decisional Diffie Hellman (DDH), Learning With Errors (LWE), N^{th} Residuosity} where the size of the shared randomness equals the output length. Recall that iO is necessary because output-compressing randomized encodings for Turing machines implies iO for circuits as shown by Lin et al.[LPST16] (it is easy to see that this implication to iO remains true even in the shared randomness model). We describe the techniques used in our construction in Section 2.1. We then use this new tool to tackle basic feasibility questions in the context of two fundamental areas in Cryptography: secure multiparty computation (MPC) and indistinguishability obfuscation (iO).

Compact MPC for Turing machines with unbounded output in the Random Oracle model. The first basic feasibility question we address is the following: Consider a set of n mutually distrusting parties with inputs $x_1, \dots, x_n$ respectively that agree on a Turing machine M. Their goal is to securely compute the output $M(x_1, \dots, x_n)$ without leaking any information about their respective inputs, where we stress that the output can be of any unbounded polynomial size. Crucially, we require that the communication complexity of the protocol (the sum of the length of the messages exchanged by all the parties) is $\mathsf{poly}(|M|, |x_1|, \dots, |x_n|, \lambda)$ for some fixed polynomial poly where λ is the security parameter. In particular, the communication complexity

[1] the size can depend logarithmically on the output length and running time.

[2] We actually consider a stronger notion where part of the input need not be hidden and we require that the size of the encoding should not grow with this revealed part. This is a generalization of the notion of partial garbling schemes introduced by Ishai and Wee [IW14].

should be independent of the output length and the running time of the machine M on input $(x_1, \ldots, x_n)$. We call such an MPC protocol to be *compact*. Indeed, this communication efficiency requirement is the most natural efficiency requirement in the context of MPC for Turing machines.

Remarkably, this extremely basic question, as stated above, has never been considered before to the best of our knowledge (see related work below for comparison with previous work). At first glance, one may think that Fully Homomorphic Encryption (FHE), one of the most powerful primitives in Cryptography, should help solve this problem. The reason being that, at least in the two party setting, FHE allows one party to encrypt its input and send it to the other party, who can then homomorphically evaluate the function to be computed "under the hood" and compute an encryption of the final output. However, it is not clear how this evaluator would learn the output since he does not have the decryption key. Sending the encryption of the final output to the other party would also blow up the communication complexity. This is related to the question posed by Hubácek and Wichs [HW15], where they consider a circuit based model, and in fact, our notion generalizes their model. That is, they consider n parties who wish to securely evaluate a circuit on their joint inputs such that the communication complexity of the protocol is independent of the output length of the circuit. In that work, they showed how to achieve semi-honest secure two party computation with this efficiency requirement assuming iO for circuits and somewhere statistically binding (SSB) hash. Further, they showed that in the context of malicious adversaries,[3] in the standard model, it is impossible to construct a secure computation protocol with such efficiency requirement even for just two party computation.

However, in this work, we are not willing to give up on achieving malicious secure compact MPC. Instead, we find a way to evade this impossibility result! We do so by considering the well studied programmable random oracle (RO) model [BR93, Nie02, DSW08, Wee09, CJS14, CDG+18]. We stress that typically, people look to the RO model in the hunt for efficiency improvements but here, we are seeking to establish basic feasibility results using the RO model. Indeed, the RO model has enabled important feasibility results in the past which were impossible in the plain model, for example unconditional non-interactive zero-knowledge arguments for NP with sub-linear communication [IMS12] and Universal Samplers [HJK+16].

More specifically, we show how to construct a *compact* constant round MPC protocol for Turing machines in the RO model secure against malicious adversaries assuming iO for circuits and any assumption in {DDH, LWE, Nth Residuosity}. Recall that by *compact*, we mean that the communication complexity of the protocol is independent of the output length and running time of the Turing machine being evaluated on the joint inputs of the parties. We obtain this result by using output-compressing randomized encodings in the shared randomness model to compile any non-compact malicious secure constant round MPC protocol (even just for circuits) in the plain model into a *compact* constant round MPC protocol for Turing machines in the RO model while preserving the round complexity. We again stress that to the best of our knowledge, this is the first MPC protocol for Turing machines where the communication complexity is bounded by a polynomial in the description length of the machine and the input lengths of all the parties. We also observe that as a corollary of our work, we obtain the first malicious secure compact MPC protocol in the circuit based model of Hubácek and Wichs [HW15], in the RO model. We describe the techniques used in our construction in Section 2.2.

Succinct iO for Turing machines for bounded inputs in the shared randomness model. The problem of bootstrapping from iO for circuits to iO for Turing machines has been the subject of intense study over the last few years. In 2015, in three concurrent works [KLW15, BGL+15, CHJV15][4] showed how to construct iO for Turing machines where the size of the obfuscation grows with a bound on the input length to the Turing machine. In this work, we ask the following question: can we construct iO for Turing machines in the shared randomness model where the obfuscator and evaluator have a shared random string that grows with the input bound but the size of the obfuscation does not?

Lin et al. [LPST16] showed that output-compressing randomized encodings are closely related to iO for Turing machines. That is, they showed that simulation secure output-compressing randomized encodings

[3]their impossibility in fact even ruled out the simpler setting of honest but deterministic adversaries - such an adversary behaves honestly in the protocol execution but fixes its random tape to some deterministic value.

[4]Recently, concurrent to our work, [AL18, AM18, GS18] also showed how to construct iO for Turing machines where, similar to [KLW15, BGL+15, CHJV15], the size of the obfuscation grows with a bound on the input length to the Turing machine.

in the plain model implies iO for Turing machines with unbounded inputs.[5] In particular, this implies iO for Turing machines with bounded inputs where the size of the obfuscation does not grow with the input bound. As we know, simulation secure output-compressing randomized encodings are impossible in the plain model. However, it turns out that this implication does not carry over in the shared randomness model. That is, if we start with output-compressing randomized encodings in the shared randomness model and apply the transformation in [LPST16], in the resulting iO scheme, the size of the obfuscation does in fact grow with the input bound. The key obstacle is that in the transformation, the obfuscation consists of an output-compressing randomized encoding that is the root of a GGM-like tree ([GGM86]). This encoding, on evaluation, outputs another output-compressing randomized encoding corresponding to its child node and the process is repeated. In order to evaluate the obfuscated program on an input of length n, the evaluator has to traverse the obfuscated program up to a depth of length n. As a result, the machine being encoded in the root needs the shared randomness for each layer, up to a depth of length n. Hence, the size of the machine encoded in the root grows with the input bound and so does the size of the obfuscated program. Note that this approach fails even if the size of the shared randomness for the encoding is just 1 bit (independent of the length of the output).

Nevertheless, we show how to overcome this obstacle by taking a completely different approach. In our solution, the obfuscated program consists of an output-compressing randomized encoding in which, crucially, neither the machine being encoded nor the input to the machine, depends on the input bound of the obfuscation scheme. Hence, the size of the encoding, and therefore, also the size of the obfuscation, does not grow with the input bound. We elaborate more about the techniques used in our construction in Section 2.3. Concretely, we obtain the following result: iO for Turing machines in the shared randomness model assuming iO for circuits and any assumption in {DDH, LWE, Nth Residuosity}, where the obfuscator and evaluator have a shared random string of length $\mathsf{poly}(n, \lambda)$ for some fixed polynomial poly, and the size of the obfuscation is $\mathsf{poly}_1(|M|, \lambda)$ for some fixed polynomial poly_1. Here, M denotes the Turing machine being obfuscated and n denotes the input bound.

1.1 Our Results

In this paper, we achieve the following results.

1) Output-compressing randomized encodings.
We prove the following theorem:

Theorem 1.1 (Informal)**.** There exists an output-compressing randomized encoding scheme for Turing machines in the shared randomness model assuming the existence of:

- iO for circuits (AND)
- A $\in$ {DDH, LWE, Nth Residuosity}.

Further, the length of the shared randomness is equal to the output length.

2) Compact MPC for Turing machines with unbounded output in the RO model.

We prove the following theorem:

Theorem 1.2 (Informal)**.** For any n, $t > 0$, there exists a constant round *compact* MPC protocol amongst n parties for Turing machines in the Programmable Random Oracle model that is malicious secure against up to t corruptions assuming the existence of:

- Output-compressing randomized encodings in the shared randomness model (AND)

[5]Lin et al. [LPST16] in fact showed that a weaker notion of distributional indistinguishability based secure output-compressing randomized encodings suffices to imply iO for Turing machines with unbounded inputs. However, they also supplement this by showing that it is impossible, in general, to construct such encodings.

- Constant round MPC protocol amongst n parties in the plain model that is malicious secure against up to t corruptions.

Once again, recall that by *compact*, we mean that the communication complexity of the protocol is independent of the output length and running time of the Turing machine being evaluated on the joint inputs of the parties. Here, we note that the above compiler even works if the underlying MPC protocol is for circuits. That is, we can convert any constant round protocol for circuits into a constant round protocol for Turing machines (with an input bound) by first converting the Turing machine into a (potentially large) circuit.

Also, we can instantiate the underlying MPC protocol in the following manner to get a round optimal compact MPC: append a non-interactive zero knowledge argument based on DLIN in the common random string model [GOS06] to either the two round semi-malicious MPC protocol of [MW16] that is based on LWE in the common random string model or the ones of [GS18, BL18] that are based on DDH/N^{th} residuosity in the plain model, to get two round malicious secure MPC protocols in the common random string model. We can then implement the common random string required for the underlying protocol via the RO. We thus achieve the following corollary:

Corollary 1.1. Assuming the existence of:

- iO for circuits (AND)
- DDH, or LWE, or Nth Residuosity (AND)
- DLIN,

there exists a *compact, round optimal* (two round) MPC protocol π for Turing machines in the Programmable Random Oracle model that is malicious secure against a dishonest majority.

Our result also gives a malicious secure *compact* MPC protocol in the circuit-based setting of [HW15] in the RO model. We also achieve other interesting corollaries by instantiating the underlying MPC protocol in the setting of super-polynomial simulation or in the setting of concurrent executions. We elaborate on both the above points in Section 6.

3) Succinct iO for Turing machines for bounded inputs in the shared randomness model.

We prove the following theorem:

Theorem 1.3 (Informal)**.** There exists an iO scheme for Turing machines in the shared randomness model where the size of the obfuscated program is independent of the input bound assuming the existence of:

- iO for circuits,
- DDH, or LWE, or Nth Residuosity.

1.2 Related work

A series of works [OS97, GHL$^+$14, GGMP16, Mia16, HY16, LO17] consider MPC for RAM programs. However, in all of them, the communication complexity of the protocol grows with the running time of the RAM program. As a result, the communication complexity of the protocol in the Turing machine model would also grow with the output length. We stress that in our work, we require that the communication complexity can grow with neither output length nor running time of the Turing machine.

Ananth et al.[AJS17] construct an iO scheme for Turing machines in which, for any machine M and input bound L, the size of the obfuscation is $|M| + \mathsf{poly}(L, \lambda)$. However, in our setting, we require that the size be independent of this bound L.

Open Problems. Hubácek and Wichs [HW15] show that some weak form of obfuscation (that does not seem to imply iO) is necessary to construct compact semi-honest secure MPC protocols in the plain model. One interesting open problem is to study whether, in fact, iO is necessary. Further, another interesting open problem is can we construct compact MPC protocols in the RO model based on weaker assumptions or in fact, obfuscation is necessary? Our construction gives an initial feasibility result for this problem and we believe it would be an interesting research direction to pursue further.

2 Technical Overview

2.1 Output Compressing Randomized Encodings

We will now discuss a high-level overview of our output-compressing randomized encoding (OcRE) scheme in the *shared randomness model*. Let $\mathcal{M}$ be a family of Turing machines with output size bounded by $\mathsf{o\text{-}len}$. An OcRE scheme for $\mathcal{M}$ in the shared randomness model consists of a setup algorithm, an encoding algorithm and a decoding algorithm. The setup algorithm takes as input security parameter λ together with a string rnd of length $\mathsf{o\text{-}len}$, and outputs a succinct encoding key ek of size $\mathsf{poly}(\lambda)$.[6] This encoding key is used by the encoding algorithm, which takes as input a machine $M \in \mathcal{M}$, an input $x \in \{0,1\}^*$, and outputs an encoding $\widetilde{M_x}$. Finally, the decoding algorithm can use $\widetilde{M_x}$ and rnd to recover $M(x)$. For efficiency, we require that the encoding time depends only on $|M|$, $|x|$ and security parameter λ. In particular, the size of the encoding should not grow with the output length $\mathsf{o\text{-}len}$ or the running time of M on x. [7]

The starting point of our construction is the succinct randomized encoding scheme of [KLW15], which is an encoding scheme for *boolean* Turing machines, and the size of the encoding depends only on $|M|, |x|$ and security parameter λ. We want to use this tool as a building block to build an encoding scheme for *general* Turing machines (i.e. with multi-bit output) where the size of the encoding still only depends on $|M|, |x|$ and λ. As a first step, let us consider the following approach. The encoding algorithm outputs an obfuscated program $\mathsf{Prog}[M, x]$, which has M and x hardwired, takes input $j \in [\mathsf{o\text{-}len}]$, and outputs a KLW encoding of M_j, x (the randomness for computing the encoding is obtained by applying a PRF on j). Here, M_j is a boolean Turing machine which, on input x, outputs the j^{th} bit of $M(x)$. The decoding algorithm runs Prog for each $j \in [\mathsf{o\text{-}len}]$, obtains $\mathsf{o\text{-}len}$ different encodings, and then decodes each of them to obtain the entire output bit by bit. Clearly, this construction satisfies the efficiency requirement. This is because the size of the program Prog depends only on $|M|, |x|$, and hence the size of the encoding only depends on $|M|, |x|, \lambda$. As far as security is concerned, it is easy to show that this scheme satisfies indistinguishability-based security; that is, if (M_0, x_0) and (M_1, x_1) are two pairs such that $M_0(x_0) = M_1(x_1)$, $|M_0| = |M_1|$, $|x_0| = |x_1|$, then the obfuscation of $\mathsf{Prog}[M_0, x_0]$ is computationally indistinguishable from the obfuscation of $\mathsf{Prog}[M_1, x_1]$. Unfortunately, recall that our goal is simulation security, and it is not possible to simulate an obfuscation of $\mathsf{Prog}[M, x]$, given only $M(x)$ as input. In particular, if $y = M(x)$ is a long pseudorandom string (whose length can be much longer than the size of $\mathsf{Prog}[M,]$), then it should be hard to compress y to a short encoding (as shown by Lin et al. [LPST16]).

As noted in the previous section, we will evade the "incompressibility" argument by allowing the shared randomness to have size that grows with the output length. Our goal will be to allow the simulator to embed the output of the machine M in this randomness. Our second attempt is as follows. The setup algorithm computes a short commitment ek to the shared randomness (say with a Merkle tree), and outputs ek as the encoding key. The encoding algorithm computes an obfuscation of $\mathsf{Prog}[M, x, \mathsf{ek}]$, which has M, x, ek hardwired, takes as input an index j, a bit b (which is supposed to be the j^{th} bit of the shared randomness), and an opening π that the bit b is indeed the j^{th} bit of the shared random string. The program checks the proof π, and then computes a KLW encoding of (M_j, x).

While the bit b is essentially ignored in the real-world encoding, it is used by the simulator in the ideal world. In the ideal world, the simulator, on receiving $M(x)$, masks it with a pseudorandomly generated

[6] We will assume $\mathsf{o\text{-}len}$ is at most 2^λ.

[7] Strictly speaking, it is allowed to depend polylogarithmically on the running time of M on input x; for this overview, we will ignore this polylogarithmic dependence on the running time.

one-time pad and outputs the resultant string as the shared randomness, and the short commitment ek is computed as in real world. For the encoding, it outputs an obfuscation of $\mathsf{Prog\text{-}sim}[\mathsf{ek}]$, which takes as input (j, b, π), checks the proof π, unmasks the bit b to obtain $M(x)_j$ and simulates the KLW randomized encoding using $M(x)_j$. This program has behavior identical to $\mathsf{Prog}[M, x, \mathsf{ek}]$ as long as the adversary only gives openings to the original bits of the shared randomness.

There is a simple problem with this idea: obfuscation only guarantees indistinguishability of programs that are functionally equivalent, and although the security of a Merkle tree would make it computationally infeasible for an adversary to come up with an opening to a wrong value, these inputs do in fact exist. To fix this problem, we use a special iO-compatible family of hash functions called 'somewhere-statistically binding (SSB) hash', introduced by [HW15]. Intuitively, this primitive is similar to a merkle tree except for two additional features. First, it allows a given position to be statistically "bound", where for that index it is only possible to give an opening for the correct bit. So there are three algorithms, $\mathsf{Setup}, \mathsf{Open}$, and Verify, as in the case of a Merkle tree, but Setup additionally takes as input a position to bind. If j is the bound position for H then there is no opening π for a bit $b \neq x_j$ such that $\mathsf{Verify}(\pi, b, j, H(x))$ accepts. Second, this bound position is hidden, so we can change it without being detected. Using this new hash allows us to make a series of hybrids where we change the shared randomness one bit at a time without giving up indistinguishability.

2.2 Compact MPC for Turing Machines in the Random Oracle Model.

We now describe the techniques used in our round preserving compiler from any non-compact constant round malicious secure MPC protocol in the plain model to a *compact* constant round malicious secure MPC protocol in the RO model, using output-compressing randomized encodings in the shared randomness model.

To begin with, consider any constant round MPC protocol π in the plain model. For simplicity, lets assume that every party broadcasts a message in each round. In order to make it *compact*, our main idea is a very simple one: use output-compressing randomized encodings to *shrink* the messages sent by every party in each round so that they are independent of the output length and running time of the machine. That is, instead of sending the actual message of protocol π, each party just sends an output-compressing randomized encoding of a machine and its private input that generates the actual message!

More precisely, consider a party P with input x that intends to send a message msg_1 in the first round as part of executing protocol π. Let's denote M to be the Turing machine that all the parties wish to evaluate. Let M_1 denote the algorithm used by the first party to generate this message msg_1 in the first round. Now, instead of sending msg_1, P sends an encoding of machine M_1 and input (x, r) where r is the randomness used by party P in protocol π. The recipient first decodes this encoding to receive P's first round message of protocol π - msg_1. Without loss of generality, let's assume that the length of randomness r is only proportional to the input length (else, internally, M_1 can apply a pseudorandom generator). In terms of efficiency, the description of the machine M_1 only depends on M and so it is easy to see that the size of the encoding does not depend on the non-compact message - msg_1. A natural initial observation is that in order to construct a simulator for the protocol, we need to generate simulated encodings. However, as we know that simulation secure output-compressing randomized encodings are impossible, we will resort to using our new encodings constructed in the shared randomness model.

Need for Random Oracle. Does this result in a compact protocol in the common random string (CRS) model, with the CRS being the shared randomness and its size grows with the output length? At this juncture, we first recall that [HW15] showed that malicious secure compact MPC is impossible even in the CRS model where the size of the CRS can grow with the output length.[8] As a result, it must be the case that our protocol is not a compact and secure MPC protocol in the CRS model. We first explain why and

[8]They actually show that it is impossible in an offline/online setting where the initial offline phase takes place independent of the parties' inputs and can have arbitrarily long communication complexity. In particular, using the offline phase to perform a coin-tossing protocol to generate the CRS implies the impossibility in the CRS model.

this also brings us to the use of the Programmable Random Oracle. To illustrate the issue, let's continue the protocol execution. Now, after receiving a message in the first round from every other party, P first decodes all these messages to compute a transcript trans for protocol π. P then computes an encoding of machine M_2 and input (x, r, trans) where M_2 is the machine used to generate the next message msg_2 and sends this in round 2. Looking ahead to the security proof, the simulator will have to generate a simulated encoding of this message and also simulate the shared randomness. To do that, the simulated shared randomness will have to depend on $M_2(x, r, \mathsf{trans})$. Notice that the simulator will have to decide the simulated shared randomness (aka the CRS) apriori *before* beginning the protocol execution. However, in the setting of malicious adversaries, this is not possible because the value trans depends on the adversary's input and randomness, both of which are not even picked before the adversary receives the CRS.

Therefore, we resort to the RO model. Now, in each round, along with its encoding, P also sends a short index. The recipient first queries the RO on this index to compute the shared randomness that is then used to decode. Looking ahead to the proof, the simulator can pick a random index that the RO has not been queried on so far and "program" the RO's output to be the simulated shared randomness. This can be executed after receiving the transcript of the previous round and before sending the pair of index and simulated encoding in any round.

Strong Output-compressing Randomized Encodings. Next, it turns out that, in fact, just standard output-compressing randomized encodings do not suffice for the above transformation. To see why, consider any round j. Let trans denote the transcript of the underlying protocol π at the end of round $(j-1)$. Now, in round j, party P sends an encoding of machine M_j and input (x, r, trans), where M_j is the machine used to generate the jth round message. However, the size of trans could depend on the output length of the protocol because trans denotes the transcript of the underlying non-compact protocol π. A natural attempt to solve would be to let trans be the transcript of the new compact protocol up to this point instead of the underlying protocol, and to let M_j decode the transcript when forming the next message. This also turns out to be problematic, though, since we now need a randomized encoding of a machine M_j which accesses the RO. As a result, since the size of the encoding in each round grows with the input to the machine being encoded, the size of the messages in each round also does depend on the output length and so, we are back to square one with a non-compact protocol!

In order to solve this issue, we make the crucial observation that the part of the input to the machine being encoded that actually grows with the output length of the protocol is actually public information. That is, we do not care about any privacy for this part of the input and only require that the size of the encoding does not grow with this public input. Corresponding to this, we define a new stronger version of output-compressing randomized encodings in the shared randomness model, which we call *strong* output-compressing randomized encodings. In more detail, the encoding algorithm takes as input a machine M, a private input x_1 and a public input x_2 and outputs an encoding. Informally, the efficiency requirement is that the size of the encoding is $\mathsf{poly}(|M|, |x_1|)$ for a fixed polynomial poly and does not depend on x_2, in addition to being independent of the output length and running time. Further, security requires that, in addition to the output $M(x_1, x_2)$, the simulator is also given the public input x_2 and the tuple of honest encoding and honest shared randomness should be indistinguishable from the tuple of simulated encoding and simulated shared randomness. Thus, if we use strong output-compressing randomized encodings, we overcome the issue. Our construction of strong output-compressing randomized encodings is very similar to the construction in Section 2.1 except that we replace the succinct randomized encodings with a stronger notion called succinct partial randomized encodings. More details can be found in Section 5.

Another subtle detail is that, while proving security, in the sequence of hybrids, it is essential that we first switch the encodings to be simulated before switching the messages of the protocol π from real to simulated. This is because we can not afford to send honest encodings of simulated messages of protocol π as the description of the simulator's machine to generate these messages could grow with the output bound. One interesting consequence of the above point is that our transformation is oblivious to whether the underlying simulator rewinds or runs in super-polynomial time. As a result, our construction naturally extends even to the setting of concurrent security if the underlying protocol is concurrently secure.

Notice that our compiler to solve this very basic feasibility question is in fact, remarkably simple, which further highlights the power of simulation secure output-compressing randomized encodings in the shared randomness model. We refer the reader to Section 6 for more details about our compact MPC protocol and proof.

2.2.1 Implication in the circuit model of [HW15]

First, recall that in the setting of Hubácek and Wichs [HW15], the goal is to construct an MPC protocol for circuits where the communication complexity is independent of the output length of the circuit. At first glance, it might seem that our construction trivially implies a result in the circuit setting as well. However, this is not quite directly true. Observe that in our protocol, the communication complexity grows with the description of the Turing machine and so, when we convert the circuit to the Turing machine model, the communication complexity grows with the size of the circuit. In the case of a circuit, the output length can in fact be proportional to the size of the circuit. To circumvent this, we will consider a Turing machine representation of a Universal circuit, that takes as input a circuit C and an input x and evaluates $C(x)$. Now, notice that the size of this universal circuit, and by extension, the size of the Turing machine evaluated, is independent of the circuit being evaluated. Further, we will set the circuit being computed - C, to be part of the "public" input to each strong output-compressing randomized encoding that is computed in each round of the protocol. Since all parties have knowledge of C, we don't need to hide this input. As a result, neither the machine being encoded nor the private input depend on the circuit being evaluated and this solves the problem. That is, the communication complexity of the resulting compiled protocol is independent of the output length of the circuit.

2.3 Succinct iO for Turing Machines in the Shared Randomness Model

We now describe the techniques used in our construction of iO for Turing machines in the shared randomness model where the size of the obfuscated program does not grow with a bound on the input length. We will denote such obfuscation schemes as *succinct* iO schemes in this section. First, we recall from the introduction that the transformation of Lin et al. [LPST16] to go from output-compressing randomized encodings to succinct iO does not work in the shared randomness model. Briefly, the reason was that if we want to support Turing machines with input length n, then there must be n chunks of the shared randomness, and the 'top-level' encoding in the LPST scheme must contain a commitment to each of the n chunks, and as a result, the size grows with n.

Therefore, our obfuscation scheme will have a completely different structure. Recall that [KLW15] showed an obfuscation scheme where the size of obfuscation of M with input bound n grows with the security parameter, input bound and machine size (but does not depend on the running time of M on any input). We will use such *weakly-succinct* obfuscation scheme to obtain succinct $i\mathcal{O}$.

Consider a program P that takes as input a Turing machine M, input bound n, and outputs a weakly-succinct obfuscation of M with input bound n (the randomness for obfuscation can be generated using a pseudorandom generator). The size of the output grows with n, size of M and security parameter λ. But the important thing to note here is that the size of program P does not grow with input bound n. Therefore, we can use output-compressing randomized encodings to construct succinct iO. The obfuscation algorithm simply outputs an encoding of program P with inputs (M, n). Clearly, the size of this encoding does not grow with n (using the efficiency property of OcRE). The proof of security follows from the security of the obfuscation scheme and the output-compressing randomized encoding scheme.

Finally, an informed reader might recall that the LPST construction required the security parameter to grow at each level, while in our case, we can work with a single security parameter. The reason for this is because their security reduction loses a factor of 2 for each level, and therefore the security parameter must grow at each level. In our case, we have a different proof structure, and the switch from encoding of P, M_0 to P, M_1 in the security proof is a single-step jump.

Organization. We first describe some preliminaries in Section 3. In section Section 4, we describe the definition of strong output-compressing randomized encodings for Turing machines in the shared randomness model and this is followed by the construction in Section 5. Then, in Section 6, we construct compact MPC protocols in the random oracle model. Our construction of succinct iO for Turing machines is described in Section 7. Finally, we describe the construction of succinct partial randomized encodings in Section 8 and we defer its proof of security to the supplementary material attached along with the submission.

3 Preliminaries

We will use λ to denote the security parameter throughout the rest of the paper. For any string s of length n, let $s[i]$ denote the i^{th} bit of s. Without loss of generality, we assume all Turing machines are oblivious.

We describe the definition of secure multiparty computation in the random oracle model in Appendix A. Some additional preliminaries can be found in Appendix B.

4 Randomized Encodings: Definitions

4.1 Succinct Partial Randomized Encodings

In this section, we introduce the notion of succinct partial randomized encodings (spRE). This is similar to the notion of succinct randomized encodings (defined in B.4), except that the adversary is allowed to learn part of the input. For efficiency, we require that if the machine has size m, and ℓ bits of input are hidden, then the size of randomized encoding should be polynomial in the security parameter λ, ℓ and m. In particular, the size of the encoding does not depend on the entire input's length (this is possible only because we want to hide ℓ bits of the input; the adversary can learn the remaining bits of the input). This notion is the Turing Machine analogue of *partial garbling* of arithmetic branching programs, studied by Ishai and Wee [IW14].

A succinct partial randomized encoding scheme SPRE for a class of boolean Turing machines $\mathcal{M}$ consists of a preprocessing algorithm $\mathsf{Preprocess}$, encoding algorithm Encode, and a decoding algorithm Decode with the following syntax.

$\mathsf{Preprocess}(1^\lambda, x_2 \in \{0,1\}^*)$: The preprocessing algorithm takes as input security parameter λ (in unary), string $y \in \{0,1\}^*$ and outputs a string hk.

$\mathsf{Encode}(M \in \mathcal{M}, T \in \mathbb{N}, x_1 \in \{0,1\}^*, \mathsf{hk} \in \{0,1\}^{p(\lambda)})$: The encoding algorithm takes as input a Turing machine $M \in \mathcal{M}$, time bound $T \in \mathbb{N}$, partial input $x_1 \in \{0,1\}^*$, string $\mathsf{hk} \in \{0,1\}^{p(\lambda)}$, and outputs an encoding $\widetilde{M}$.

$\mathsf{Decode}(\widetilde{M}, x_2, \mathsf{hk})$: The decoding algorithm takes as input an encoding $\widetilde{M}$, a string $x_2 \in \{0,1\}^*$, string hk and outputs $y \in \{0,1,\perp\}$.

Definition 4.1. Let $\mathcal{M}$ be a family of Turing machines. A randomized encoding scheme $\mathsf{SPRE} = (\mathsf{Preprocess}, \mathsf{Encode}, \mathsf{Decode})$ is said to be a succinct partial randomized encoding scheme if it satisfies the following correctness, efficiency and security properties.

- Correctness: For every machine $M \in \mathcal{M}$, string $x = (x_1, x_2) \in \{0,1\}^*$, security parameter λ and $T \in \mathbb{N}$, if $\mathsf{hk} \leftarrow \mathsf{Preprocess}(1^\lambda, x_2)$, then $\mathsf{Decode}(\mathsf{Encode}(M, T, x_1, \mathsf{hk}), x_2) = \mathsf{TM}(M, x, T)$.

- Efficiency: There exist polynomials $p_{\mathsf{prep}}, p_{\mathrm{enc}}$ and p_{dec} such that for every machine $M \in \mathcal{M}$, $x = (x_1, x_2) \in \{0,1\}^*$, $T \in \mathbb{N}$ and $\lambda \in \mathbb{N}$, if $\mathsf{hk} \leftarrow \mathsf{Preprocess}(1^\lambda, x_2)$, then $|\mathsf{hk}| = p_{\mathsf{prep}}(\lambda)$, the time to encode $\widetilde{M} \leftarrow \mathsf{Encode}(M, T, x_1, \mathsf{hk})$ is bounded by $p_{\mathrm{enc}}(|M|, |x_1|, \log T, \lambda)$, and the time to decode $\widetilde{M}$ is bounded by $\min(\mathsf{Time}(M, x, T) \cdot p_{\mathrm{dec}}(\lambda, \log T)$.
- Security: For every PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a PPT simulator $\mathcal{S}$ such that for all PPT distinguishers $\mathcal{D}$, there exists a negligible function $\mathrm{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$, $\Pr[1 \leftarrow \mathcal{D}(\mathsf{Expt\text{-}SPRE\text{-}Real}_{\mathsf{SPRE},\mathcal{A}}(\lambda))] - \Pr[1 \leftarrow \mathcal{D}(\mathsf{Expt\text{-}SPRE\text{-}Ideal}_{\mathsf{SRE},\mathcal{A},\mathcal{S}}(\lambda))] \leq \mathrm{negl}(\lambda)$, where $\mathsf{Expt\text{-}SPRE\text{-}Real}$ and $\mathsf{Expt\text{-}SPRE\text{-}Ideal}$ are defined in Figure 11. Moreover, the running time of $\mathcal{S}$ is bounded by some polynomial $p_{\mathcal{S}}(|M|, |x_1|, \log T, \lambda)$.

Experiments $\mathsf{Expt\text{-}SPRE\text{-}Real}_{\mathsf{SPRE},\mathcal{A}}(\lambda)$ **and** $\mathsf{Expt\text{-}SPRE\text{-}Ideal}_{\mathsf{SPRE},\mathcal{A},\mathcal{S}}(\lambda)$

$\mathsf{Expt\text{-}SPRE\text{-}Real}_{\mathsf{SPRE},\mathcal{A}}(\lambda)$:

- $(M, x = (x_1, x_2), T, \sigma) \leftarrow \mathcal{A}_1(1^\lambda)$.
- $\mathsf{hk} \leftarrow \mathsf{Preprocess}(x_2, 1^\lambda)$.
- $\widetilde{M} \leftarrow \mathsf{Encode}(M, T, x_1, \mathsf{hk})$.
- Experiment outputs $\mathcal{A}_2(\widetilde{M}, \sigma)$.

$\mathsf{Expt\text{-}SPRE\text{-}Ideal}_{\mathsf{SPRE},\mathcal{A},\mathcal{S}}(\lambda)$:

- $(M, x = (x_1, x_2), T, \sigma) \leftarrow \mathcal{A}_1(1^\lambda)$,
  $t^* = \min(T, \mathsf{Time}(M, x))$, $\mathrm{out} = \mathsf{TM}(M, x, T)$.
- $\mathsf{hk} \leftarrow \mathsf{Preprocess}(1^\lambda, x_2)$.
- $\widetilde{M} \leftarrow \mathcal{S}\left(1^{|M|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, \mathrm{out}, t^*\right)$.
- Experiment outputs $\mathcal{A}_2(\widetilde{M}, \sigma)$.

Figure 1: Simulation Security Experiments for partial randomized encodings

Our construction of succinct partial randomized encodings is closely related to the succinct randomized encodings scheme by [KLW15] and we defer the details to Appendix 8.

4.2 Strong output-compressing Randomized Encodings in the shared randomess model

The notion of succinct randomized encodings (defined in Appendix B) was originally defined for boolean Turing machines. We can also consider randomized encodings for Turing machines with long outputs. Using (standard) succinct randomized encodings, one can construct randomized encodings for Turing machines with multi-bit outputs, where the size of encodings grows linearly with the output size. In a recent work, Lin et al. [LPST16] introduced a stronger notion called *output-compressing randomized encodings*, where the size of the encoding only depends sublinearly on the output length. Lin et al. also showed that simulation based security notions of output-compressing randomized encodings are impossible to achieve. In this work, we consider a stronger notion of output-compressing randomized encodings in the shared randomness model where the encoder and decoder have access to a shared random string (denoted by crs). Here, the machine also takes another public input x_2 along with a private input x_1 with the requirement that the size of the encoding should only grow polynomially in the size of the machine and the private input x_1. In particular, it does not grow with x_2 or the running time of the machine or its output length. We define it formally below.

A strong output-compressing randomized encoding scheme $\mathsf{S.OcRE} = (\mathsf{Setup}, \mathsf{Encode}, \mathsf{Decode})$ in the shared randomness model consists of three algorithms with the following syntax.

$\mathsf{Setup}(1^\lambda, 1^{\mathsf{o\text{-}len}}, \mathsf{crs} \in \{0,1\}^{\mathsf{o\text{-}len}})$: The setup algorithm takes as input security parameter λ, output-bound $\mathsf{o\text{-}len}$ and a shared random string crs of length $\mathsf{o\text{-}len}$. It outputs an encoding key ek.

$\mathsf{Encode}((M, \mathsf{tmf}(\cdot)), x = (x_1, x_2), T, \mathsf{ek})$: The encoding algorithm takes as input an oblivious Turing Machine M with tape movement function $\mathsf{tmf}(\cdot)$, input x consisting of a private part x_1 and a public part x_2, time bound $T \leq 2^\lambda$ (in binary) and an encoding key ek, and outputs an encoding $\widetilde{M_x}$.

$\mathsf{Decode}(\widetilde{M_x}, x_2, \mathsf{crs})$: The decoding algorithm takes as input an encoding $\widetilde{M_x}$, a public input x_2, the shared random string crs and outputs $y \in \{0,1\}^* \cup \{\perp\}$.

Definition 4.2. A strong output-compressing randomized encoding scheme $\mathsf{S.OcRE} = (\mathsf{Setup}, \mathsf{Encode}, \mathsf{Decode})$ in the shared randomness model is said to be secure if it satisfies the following correctness, efficiency and security requirements.

- Correctness: For all security parameters $\lambda \in \mathbb{N}$, output-length bound $\mathsf{o\text{-}len} \in \mathbb{N}$, $\mathsf{crs} \in \{0,1\}^{\mathsf{o\text{-}len}}$, machine M with tape movement function $\mathsf{tmf}(\cdot)$, input $x = (x_1, x_2)$, time bound T such that $|M(x)| \leq \mathsf{o\text{-}len}$, if $\mathsf{ek} \leftarrow \mathsf{Setup}(1^\lambda, 1^{\mathsf{o\text{-}len}}, \mathsf{crs})$, $\widetilde{M_x} \leftarrow \mathsf{Encode}((M, \mathsf{tmf}(\cdot)), x, T\mathsf{ek})$, then $\mathsf{Decode}(\widetilde{M_x}, x_2, \mathsf{crs}) = \mathsf{TM}(M, x, T)$.

- Efficiency: There exist polynomials p_1, p_2, p_3 such that for all $\lambda \in \mathbb{N}$, $\mathsf{o\text{-}len} \in \mathbb{N}$, $\mathsf{crs} \in \{0,1\}^{\mathsf{o\text{-}len}}$:
  1. If $\mathsf{ek} \leftarrow \mathsf{Setup}(1^\lambda, 1^o, \mathsf{crs})$, $|\mathsf{ek}| \leq p_1(\lambda, \log o)$.
  2. For every Turing machine M, time bound T, input $x = (x_1, x_2) \in \{0,1\}^*$, if $\widetilde{M_x} \leftarrow \mathsf{Encode}(M, x, T, \mathsf{ek})$, then $|\widetilde{M_x}| \leq p_2(|M|, |x_1|, \log |x_2|, \log T, \log o, \lambda)$.
  3. The running time of $\mathsf{Decode}(\widetilde{M_x}, x_2, \mathsf{crs})$ is at most $\min(T, \mathsf{Time}(M, x)) \cdot p_3(\lambda, \log T)$.

- Security: For every PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a simulator $\mathcal{S}$ such that for all PPT distinguishers $\mathcal{D}$, there exists a negligible function $\mathrm{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\begin{aligned}&\Pr[1 \leftarrow \mathcal{D}(\mathsf{Expt\text{-}S.OcRE\text{-}Real}_{\mathsf{S.OcRE},\mathcal{A}}(\lambda))] \\ &- \Pr[1 \leftarrow \mathcal{D}(\mathsf{Expt\text{-}S.OcRE\text{-}Ideal}_{\mathsf{S.OcRE},\mathcal{A},\mathcal{S}}(\lambda))] \leq \mathrm{negl}(\lambda),\end{aligned}$$

  where $\mathsf{Expt\text{-}S.OcRE\text{-}Real}$ and $\mathsf{Expt\text{-}S.OcRE\text{-}Ideal}$ are defined in Figure 2.

Experiments $\mathsf{Expt\text{-}S.OcRE\text{-}Real}_{\mathsf{S.OcRE},\mathcal{A}}(\lambda)$ **and** $\mathsf{Expt\text{-}S.OcRE\text{-}Ideal}_{\mathsf{S.OcRE},\mathcal{A},\mathcal{S}}(\lambda)$

$\mathsf{Expt\text{-}S.OcRE\text{-}Real}_{\mathsf{S.OcRE},\mathcal{A}}(\lambda)$:

- $(1^{\mathsf{o\text{-}len}}, (M, \mathsf{tmf}(\cdot)), x = (x_1, x_2), T, \sigma) \leftarrow \mathcal{A}_1(1^\lambda)$.
- $\mathsf{crs} \leftarrow \{0,1\}^{\mathsf{o\text{-}len}}$, $\mathsf{ek} \leftarrow \mathsf{Setup}(1^\lambda, 1^{\mathsf{o\text{-}len}}, \mathsf{crs})$.
- $\widetilde{M} \leftarrow \mathsf{Encode}((M, \mathsf{tmf}(\cdot)), x, T, \mathsf{ek})$.
- Experiment outputs $\mathcal{A}_2(\mathsf{crs}, \mathsf{ek}, \widetilde{M}, \sigma)$.

$\mathsf{Expt\text{-}S.OcRE\text{-}Ideal}_{\mathsf{S.OcRE},\mathcal{A}}(\lambda)$:

- $(1^{\mathsf{o\text{-}len}}, (M, \mathsf{tmf}(\cdot)), x = (x_1, x_2), T, \sigma) \leftarrow \mathcal{A}_1(1^\lambda)$.
- Let $t^* = \min(T, \mathsf{Time}(M, x))$ and $b^* = \mathsf{TM}(M, x, T)$.
- $s \leftarrow \mathcal{S}(1^{|M|}, 1^{|x_1|}, \mathsf{tmf}(\cdot), x_2, t^*, b^*, 1^\lambda)$.
- Let $s = (\mathsf{crs}, \widetilde{M})$.
- $\mathsf{ek} \leftarrow \mathsf{Setup}(1^\lambda, 1^{\mathsf{o\text{-}len}}, \mathsf{crs})$.
- Experiment outputs $\mathcal{A}_2(\mathsf{crs}, \mathsf{ek}, \widetilde{M}, \sigma)$.

Figure 2: Simulation Security Experiments for strong output-compressing randomized encodings in the shared randomness model

Remark: In particular, note that strong output-compressing randomized encodings ($\mathsf{S.OcRE}$) implies output-compressing randomized encodings (OcRE) by setting the public input x_2 to be $\perp$.

5 Strong Output-compressing Randomized Encodings in the CRS Model

In this section, we show a construction of strong output-compressing randomized encodings in the common random string (CRS) model. Formally, we show the following theorem:

Theorem 5.1. Assuming the existence of:

- iO for circuits (AND)
- Somewhere statistically binding (SSB) hash (AND)

- Puncturable PRFs (AND)
- Succinct partial randomized encodings for single-bit output Turing machines,

There exists a strong output-compressing randomized encoding scheme for Turing machines in the shared randomness model.

Instantiating the SSB hash and the succinct partial randomized encodings, we get the following corollary:

Corollary 5.1. Assuming the existence of:

- iO for circuits (AND)
- A $\in$ {DDH, LWE, Nth Residuosity},

There exists a strong output-compressing randomized encoding scheme for Turing machines in the shared randomness model.

Notation and Primitives used: We will be using the following cryptographic primitives for our construction:

- Indistinguishability obfuscation for circuits ($\mathsf{Ckt.Obf}$, $\mathsf{Ckt.Eval}$).
- Succinct partial randomized encodings for single-bit output Turing machines ($\mathsf{SPRE.Preprocess}$, $\mathsf{SPRE.Encode}$, $\mathsf{SPRE.Decode}$). Without loss of generality, we assume that the algorithm $\mathsf{SPRE.Encode}$ uses λ bits of randomness - it can internally apply a PRG on this randomness if a larger amount is required.
- Somewhere statistically binding hash($\mathsf{SSB.Gen}$, $\mathsf{SSB.Open}$, $\mathsf{SSB.Verify}$).
- A Puncturable PRF (F_1, $\mathsf{PPRF.Puncture}_1$) that takes inputs of size λ and outputs 1 bit.
- A Puncturable PRF (F_2, $\mathsf{PPRF.Puncture}_2$) that takes inputs of size λ and outputs λ bits.

5.1 Construction

$\mathsf{S.OcRE.Setup}(1^\lambda, 1^o, \mathsf{crs} \in \{0,1\}^o)$: The setup algorithm does the following:

1. Choose hash function $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, 0)$.[9]
2. Compute $h = H(\mathsf{crs})$ and set $\mathsf{ek} = (h, H)$.

$\mathsf{S.OcRE.Encode}(M, x = (x_1, x_2), T, \mathsf{ek} = (h, H))$: The encoding algorithm does the following:

1. Compute $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$.
2. Choose a key K_{SPRE} for the puncturable PRF F_2.
3. Let M_i denote the turing machine that, on input x, runs the machine M on input x and outputs the i^{th} bit of $M(x)$. Let t denote $|\mathsf{SPRE.Encode}(M_i, T, x_1, \mathsf{hk}; r)|$ using any random string r.
4. Compute $\widetilde{\mathsf{Prog}} \leftarrow \mathsf{Ckt.Obf}(\mathsf{Prog}, 1^\lambda)$ where the program Prog is defined in Figure 3. Note that the size of the program Prog is padded appropriately so that it is equal to the size of the program $\mathsf{Prog\text{-}sim}$ defined later in Figure 4.
5. Output $\widetilde{M_x} = (\widetilde{\mathsf{Prog}}, t, H)$.

[9]We modify the syntax of the SSB hash system slightly to allow the binding index to range from $0, \ldots, o$ and without loss of generality, just set $\mathsf{SSB.Gen}(1^\lambda, o, 0) = \mathsf{SSB.Gen}(1^\lambda, o, 1)$. That is, when the binding index is set as 0, we actually don't care at what index the hash system is bound at and will not actually use the statistically binding property. This is just to be consistent with the definition of the SSB hash system.

Prog

Inputs : Index $i \in [o]$, bit $\mathsf{str} \in \{0,1\}$, proof π, index $j \in [t]$.
Hardwired : Hash value $h \in \{0,1\}^\lambda$, machine M, input x_1, PRF key K_{SPRE}, bound T, preprocessed value hk.

1. Verify proof π : Check if $\mathsf{SSB.Verify}(H, h, i, \mathsf{str}, \pi) = 1$. If not, output $\perp$.
2. Recall that M_i denotes the turing machine that, on input x, runs the machine M on input x and outputs the i^{th} bit of $M(x)$. Compute out $= \mathsf{SPRE.Encode}(M_i, T, x_1, \mathsf{hk}; F_2(K_{\mathsf{SPRE}}, i))$ and output the j^{th} bit of out.

Figure 3: Circuit Prog

$\mathsf{S.OcRE.Decode}(\widetilde{M_x} = (\widetilde{\mathsf{Prog}}, t, H), x_2, \mathsf{crs})$: For each $i \in [o]$, the decoding algorithm computes bit out$_i$ as follows:

1. Parse $\mathsf{crs} = (\mathsf{crs}[1], \mathsf{crs}[2], \ldots, \mathsf{crs}[o])$, where each $\mathsf{crs}[j]$ is a bit.
2. Compute SSB proof for each $\mathsf{crs}[j]$; that is, compute $\pi[j] = \mathsf{SSB.Open}(H, \mathsf{crs}, j)$.
3. For $j = 1$ to t, do:
   (a) Compute $\widetilde{M_i}[j] = \mathsf{Ckt.Eval}(\widetilde{\mathsf{Prog}}, (i, \mathsf{crs}[i], \pi[i], j))$.
4. Let $\widetilde{M_i} = (\widetilde{M_i}[1]\ \widetilde{M_i}[2]\ \ldots\ \widetilde{M_i}[t])$. Compute out$_i = \mathsf{SPRE.Decode}(\widetilde{M_i}, x_2)$.

Finally, it outputs (out$_1$ out$_2$ $\ldots$ out$_o$).

Correctness and Succinctness Correctness follows from the correctness of (SPRE.Encode,SPRE.Decode) and (Ckt.Obf, Ckt.Eval).

Below we show the three efficiency properties required by the definition.

1. If $\mathsf{ek} \leftarrow \mathsf{Setup}(1^\lambda, 1^o, \mathsf{crs})$, $|\mathsf{ek}| = \ell_{\text{hash}}(\lambda) + \ell_{\text{fn}}(\lambda)$, where ℓ_{hash} and ℓ_{fn} are from SSB.
2. For every Turing machine M, time bound T, input $x = (x_1, x_2) \in \{0,1\}^*$, if $\widetilde{M_x} \leftarrow \mathsf{Encode}(M, x, T, \mathsf{ek})$, then $|\widetilde{M_x}| = (|prog| + |t|) \leq |\mathsf{Prog}| + \mathsf{poly}(\lambda)$. Prog is padded to be the same length as the programs used in the hybrids and Prog-sim, so $|\mathsf{Prog}|$ is the maximum of the length of these programs. By inspecting the values hardwired in each of these programs we get $|\mathsf{Prog}| \leq p(|h|, |M|, |x_1|, |hk|, k, \log o, t)$, where k is the maximum size of the keys of F_1 and F_2. By the efficiency of SPRE, the definition of SSB hashes and the definition of puncturable PRFs we get that $|\mathsf{Prog}| \leq p_2(\lambda, |M|, |x_1|, \log o)$ and thus $|\widetilde{M_x}| \leq p_2(\lambda, |M|, |x_1|, \log |x_2|, \log o)$ for some fixed polynomial p_2.
3. The running time of $\mathsf{Decode}(\widetilde{M_x}, x_2, \mathsf{crs})$ is at most $O(o \times t_1 + o \times t \times t_2)$ where t_1 is the running time of $\mathsf{SPRE.Decode}(\widetilde{M_i}, x_2)$ and t_2 is the running time of $\mathsf{Ckt.Eval}(\widetilde{\mathsf{Prog}}, (i, \mathsf{crs}[i], \pi[i], j))$. By the efficiency of the SPRE scheme and the iO scheme we have $\mathsf{Decode}(\widetilde{M_x}, x_2, \mathsf{crs}) \leq \min(T, \mathsf{Time}(M, x)) \cdot p_3(\lambda, \log T)$.

5.2 Proof of Security

5.2.1 Description of Simulator

The simulator S.OcRE.Sim gets as input the value $M(x)$ (which is the output of the machine M on input x) and the public part of the input x_2, and it must simulate the shared random string crs and an encoding $\widetilde{M_x}$ of the machine M and x. We now describe the simulator.

$\mathbf{S.OcRE.Sim(1^{|M|}, 1^{|x_1|}, x_2, 1^\lambda, M(x), T)}$:
The simulator does the following:

1. Compute $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$.

2. Choose a key K_{crs} for the puncturable PRF F_1 and a key K_{sim} for the puncturable PRF F_2.

3. Then, for each i, compute $\mathsf{crs}[i] = M(x)_i \oplus w^i$ where $w^i = F(K_{\mathsf{crs}}, i)$ and $M(x)_i$ denotes the i^{th} bit of $M(x)$. The shared random string is set to be $(\mathsf{crs}[1]\ \mathsf{crs}[2]\ \ldots\ \mathsf{crs}[o])$.

4. Choose a hash function $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, 0)$ and compute $h = H(\mathsf{crs})$.

5. Compute $\widetilde{\mathsf{Prog\text{-}sim}} \leftarrow \mathsf{Ckt.Obf}(\mathsf{Prog\text{-}sim}, 1^\lambda)$, where $\mathsf{Prog\text{-}sim}$ is defined in Figure 4.

6. Let M_i denote the turing machine that, on input x, runs the machine M on input x and outputs the i^{th} bit of $M(x)$. Let t denote $|\mathsf{SPRE.Encode}(M_i, T, z, \mathsf{hk}; r)|$) using any random string r and any input z such that $|z| = |x_1|$.

7. Set $\widetilde{M_x} = (\widetilde{\mathsf{Prog\text{-}sim}}, t)$.

Prog-sim

Inputs : Index $i \in [o]$, bit $\mathsf{str} \in \{0,1\}$, proof π, index $j \in [t]$
Hardwired : Hash $h \in \{0,1\}^\lambda$, machine M, PRF keys $K_{\mathrm{sim}}, K_{\mathsf{crs}}$, preprocessed value hk.

1. Verify proof π : Check if $\mathsf{SSB.Verify}(H, h, i, \mathsf{str}, \pi) = 1$. If not, output $\perp$.
2. Do the following:
   (a) Let $w = F_1(K_{\mathsf{crs}}, i)$ and $y = w \oplus \mathsf{str}$.
   (b) Compute out $= \mathsf{SPRE.Sim}(1^{|M_i|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, y, T; r)$ where $r = F_2(K_{\mathrm{sim}}, i)$.
   (c) Output j^{th} bit of out.

Figure 4: Simulated Program Prog-sim

5.2.2 Hybrids

We will show that the real and ideal worlds are indistinguishable via a sequence of $(o+2)$ hybrid experiments Hyb_0 to Hyb_{o+1} where Hyb_0 corresponds to the real world and Hyb_{o+1} corresponds to the ideal world. For each $i \in [o]$, in hybrid Hyb_{i^*}, the first i^* bits of the CRS are computed as encryptions of output bits (with the w's as one time pads). The encoding of M, x does not compute the SRE for $i \leq i^*$. More formally:

Hybrid Hyb_{i^*}:
The challenger does the following:

1. Compute $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$.

2. Choose a key K_{crs} for the puncturable PRF F_1 and two keys $K_{\mathrm{sim}}, K_{\mathsf{SPRE}}$ for the puncturable PRF F_2.

3. Then, for each $i \leq i^*$, compute $\mathsf{crs}[i] = M(x)_i \oplus w^i$ where $w^i = F_1(K_{\mathsf{crs}}, i)$ and $M(x)_i$ denotes the i^{th} bit of $M(x)$.

4. For each $i > i^*$, pick $\mathsf{crs}[i]$ uniformly at random.

5. The shared random string is set to be $(\mathsf{crs}[1]\ \mathsf{crs}[2]\ \ldots\ \mathsf{crs}[o])$.

6. Choose a hash function $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^*)$ and compute $h = H(\mathsf{crs})$. Set $\mathsf{ek} = h$.

7. Compute $\widetilde{\mathsf{Prog\text{-}}i^*} \leftarrow \mathsf{Ckt.Obf}(\mathsf{Prog\text{-}}i^*, 1^\lambda)$, where $\mathsf{Prog\text{-}}i^*$ is defined in Figure 5.

8. Let M_i denote the turing machine that, on input x, runs the machine M on input x and outputs the i^{th} bit of $M(x)$. Let t denote $|\mathsf{SPRE.Encode}(M_i, T, x_1, \mathsf{hk}; r)|$ using any random string r.

```
Prog-$i^*$

Inputs : Index $i \in [o]$, bit $\mathsf{str} \in \{0,1\}$, proof $\pi$, index $j \in [t]$
Hardwired : Index $i^*$, Hash $h \in \{0,1\}^\lambda$, machine $M$, input $x_1$, PRF keys $K_{\mathsf{crs}}, K_{\mathrm{sim}}, K_{\mathsf{SPRE}}$, bound $T$, preprocessed value $\mathsf{hk}$.

  1. Verify proof $\pi$ : Check if $\mathsf{SSB.Verify}(H, h, i, \mathsf{str}, \pi) = 1$. If not, output $\perp$.
  2. If $i \leq i^*$, do the following:
     (a) Let $w = F_1(K_{\mathsf{crs}}, i)$ and $y = w \oplus \mathsf{str}$.
     (b) Compute out $= \mathsf{SPRE.Sim}(1^{|M_i|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, y, T; r)$ where $r = F_2(K_{\mathrm{sim}}, i)$.
     (c) Output $j^{th}$ bit of out.
  3. Else, if $i > i^*$: Recall that $M_i$ denotes the turing machine that, on input $x$, runs the machine $M$ on input $x$ and outputs the $i^{th}$ bit of $M(x)$. Compute out $= \mathsf{SPRE.Encode}(M_i, T, x_1, \mathsf{hk}; F_2(K_{\mathsf{SPRE}}, i))$ and output the $j^{th}$ bit of out.
```

Figure 5: Hybrid Program Prog-i^*

9. Set $\widetilde{M_x} = (\widetilde{\mathsf{Prog}\text{-}i^*}, t)$.

Hybrid Hyb_{o+1}:
Identical to Hyb_o except that the value x_1 is not hardwired into Prog-i^*.

5.2.3 Indistinguishability of Hybrids

We will now show that every pair of consecutive hybrids is computationally indistinguishable and this completes the proof. Formally, we will prove the following theorem. Note that Hyb_{o+1} is indistinguishable from Hyb_o by the security of the iO scheme.

Theorem 5.2. For any index $i^* \in [o]$, the hybrids Hyb_{i^*} and Hyb_{i^*+1} are computationally indistinguishable.

Proof. We will prove this theorem via a sequence of sub-hybrids H_0 to H_9 where H_0 corresponds to Hyb_{i^*} and H_9 corresponds to Hyb_{i*+1}. Note that we will drop the term t from the description of the encoding in the rest of the proof since it is the same value throughout.

Hybrid H_0 This sub-hybrid corresponds to Hyb_{i^*}, where the adversary receives an obfuscation of Prog-i^* and a hybrid shared random string. The first i^* components of crs consists of encryptions of the first i^* bits of $M(x)_{i^*}$ using the respective w's as one time pads. The remaining components are chosen uniformly at random.

1. The adversary sends $M, x = (x_1, x_2), 1^o, T$ to the challenger.
2. The challenger computes $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$.
3. Then, the challenger chooses a key K_{crs} for the puncturable PRF F_1 and keys $K_{\mathrm{sim}}, K_{\mathsf{SPRE}}$ for the puncturable PRF F_2.
4. Next, the challenger computes the crs bits using K_{crs} for each $i \leq i^*$. That is, it sets $\mathsf{crs}[i] = M(x)_i \oplus F_1(K_{\mathsf{crs}}, i)$ for each $i \in [i^*]$. For all $i > i^*$, the challenger chooses $\mathsf{crs}[i] \leftarrow \{0,1\}$.
5. The challenger then chooses hash function $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^*)$. It computes $h = H(\mathsf{crs})$.
6. Finally, the challenger computes an obfuscation of Prog-i^*. It computes $\widetilde{M_x} \leftarrow \mathsf{Ckt.Obf}(\mathsf{Prog}\text{-}i^*, 1^\lambda)$ and sends $(\mathsf{crs}, \widetilde{M_x})$ to the adversary.
7. The adversary sends its guess b.

Hybrid H_1 In this hybrid, the challenger uses $\mathsf{SSB.Gen}$ to be binding at position i^*+1 to the bit $\mathsf{crs}[i^*+1]$.

1. The adversary sends $M, x = (x_1, x_2), 1^o, T$ to the challenger.
2. Then, the challenger computes $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$.
3. The challenger chooses a key K_{crs} for the puncturable PRF F_1 and keys $K_{\mathrm{sim}}, K_{\mathsf{SPRE}}$ for the puncturable PRF F_2.
4. Next, the challenger computes the crs bits using K_{crs} for each $i \leq i^*$. That is, it sets $\mathsf{crs}[i] = M(x)_i \oplus F_1(K_{\mathsf{crs}}, i)$ for each $i \in [i^*]$. For all $i > i^*$, the challenger chooses $\mathsf{crs}[i] \leftarrow \{0,1\}$.
5. The challenger then chooses hash function $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^*+1)$. It computes $h = H(\mathsf{crs})$.
6. Finally, the challenger computes an obfuscation of $\mathsf{Prog}\text{-}i^*$. It computes $\widetilde{M_x} \leftarrow \mathsf{Ckt.Obf}(\mathsf{Prog}\text{-}i^*, 1^\lambda)$ and sends $(\mathsf{crs}, \widetilde{M_x})$ to the adversary.
7. The adversary sends its guess b.

Hybrid H_2 In this hybrid, the adversay receives an obfuscation of $\mathsf{Prog}\text{-}(i^*, 1)$ (defined in Figure 6). This program is similar to $\mathsf{Prog}\text{-}i^*$, except that the keys K_{sim} and K_{SPRE} are punctured at (i^*+1). The challenger hardwires the output for index (i^*+1).

1. The adversary sends $M, x = (x_1, x_2)$ to the challenger.
2. The challenger computes $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$.
3. Then, the challenger chooses a key K_{crs} for the puncturable PRF F_1 and keys $K_{\mathrm{sim}}, K_{\mathsf{SPRE}}$ for the puncturable PRF F_2.
4. Next, the challenger computes the crs blocks using K_{crs} for each $i \leq i^*$. It sets $\mathsf{crs}[i] = M(x)_i \oplus F_1(K_{\mathsf{crs}}, i)$ for each $i \in [i^*]$. For all $i > i^*$, the challenger chooses $\mathsf{crs}[i] \leftarrow \{0,1\}$.
5. The challenger then chooses hash function $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^*+1)$. It computes $h = H(\mathsf{crs})$.
6. Finally, the challenger computes an obfuscation of $\mathsf{Prog}\text{-}(i^*, 1)$ defined in Figure 6. It performs the following steps.
   (a) It first computes punctured keys $K'_{\mathsf{SPRE}} = K_{\mathsf{SPRE}}\{i^*+1\} \leftarrow \mathsf{PPRF.Puncture}_2(K_{\mathsf{SPRE}}, i^*+1)$, $K'_{\mathrm{sim}} = K_{\mathrm{sim}}\{i^*+1\} \leftarrow \mathsf{PPRF.Puncture}_2(K_{\mathrm{sim}}, i^*+1)$.
   (b) Next, it computes an encoding y_1 of M_{i^*+1}, x with randomness $F_2(K_{\mathsf{SPRE}}, i^*+1)$. That is, $y_1 = \mathsf{SPRE.Encode}(M_{i^*+1}, T, x_1, \mathsf{hk}; F_2(K_{\mathsf{SPRE}}, i^*+1))$.
   (c) It computes $\widetilde{M_x} \leftarrow \mathsf{Ckt.Obf}(\mathsf{Prog}\text{-}(i^*, 1)[i^*, h, M, x_1, K_{\mathsf{crs}}, K'_{\mathrm{sim}}, K'_{\mathsf{SPRE}}, \mathsf{crs}[i^*+1], y_1, \mathsf{hk}], 1^\lambda)$ and sends $(\mathsf{crs}, \widetilde{M_x})$ to the adversary.
7. The adversary sends its guess b.

```
Prog-$(i^*, 1)$

Inputs : Index $i \in [o]$, bit $\mathsf{str} \in \{0,1\}$, proof $\pi$, index $j \in [t]$
Hardwired : Index $i^*$, Hash $h \in \{0,1\}^\lambda$, machine $M$, input $x_1$, $K_{\mathsf{crs}}$, punctured PRF keys $K'_{\mathsf{crs}}, K'_{\mathrm{sim}}, K'_{\mathsf{SPRE}}$, bit $\mathsf{crs}[i^*+1]$, output $y_1$, preprocessed value $\mathsf{hk}$, bound $T$.

  1. Verify proof $\pi$ : Check if $\mathsf{SSB.Verify}(H, h, i, \mathsf{str}, \pi) = 1$. If not, output $\perp$.
  2. If $i \leq i^*$, do the following:
      (a) Let $w = F_1(K_{\mathsf{crs}}, i)$ and $y$ be the first bit of $w \oplus \mathsf{str}$.
      (b) Compute out $= \mathsf{SPRE.Sim}(1^{|M_i|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, y, T; r)$ where $r = F_2(K'_{\mathrm{sim}}, i)$.
      (c) Output $j^{th}$ bit of out.
  3. If $i = i^* + 1$ and $\mathsf{str} = \mathsf{crs}[i^*+1]$ then output $j^{th}$ bit of $y_1$.
  4. Else, if $i > i^*$: Recall that $M_i$ denotes the turing machine that, on input $x$, runs the machine $M$ on input $x$ and outputs the $i^{th}$ bit of $M(x)$. Compute out $= \mathsf{SPRE.Encode}(M_i, T, x_1, \mathsf{hk}; F_2(K_{\mathsf{SPRE}}, i))$ and output the $j^{th}$ bit of out.
```
Figure 6: Program Prog-$(i^*, 1)$ used in sub-hybrid H_2

Hybrid H_3 This sub-hybrid is similar to the previous one, except that the hardwired randomized encoding is generated using true randomness.

1. The adversary sends $M, x = (x_1, x_2), 1^o, T$ to the challenger.
2. The challenger computes $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$.
3. Then, the challenger chooses a key K_{crs} for the puncturable PRF F_1 and keys $K_{\mathrm{sim}}, K_{\mathsf{SPRE}}$ for the puncturable PRF F_2.
4. Next, the challenger computes the crs bits using K_{crs} for each $i \leq i^*$. It sets $\mathsf{crs}[i] = M(x)_i \oplus F_1(K_{\mathsf{crs}}, i)$ for each $i \in [i^*]$. For all $i > i^*$, the challenger chooses $\mathsf{crs}[i] \leftarrow \{0,1\}$.
5. The challenger then chooses hash function $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^*+1)$. It computes $h = H(\mathsf{crs})$.
6. Finally, the challenger computes an obfuscation of Prog-$(i^*, 1)$. It performs the following steps.
   (a) It first computes punctured keys $K'_{\mathsf{SPRE}} = K_{\mathsf{SPRE}}\{i^*+1\} \leftarrow \mathsf{PPRF.Puncture}_2(K_{\mathsf{SPRE}}, i^*+1)$, $K'_{\mathrm{sim}} = K_{\mathrm{sim}}\{i^*+1\} \leftarrow \mathsf{PPRF.Puncture}_2(K_{\mathrm{sim}}, i^*+1)$.
   (b) Next, it computes an encoding y_1 of $(M_{i^*+1}, x$ with randomness $r_1 \leftarrow \{0,1\}^{\ell_{\mathrm{sre\text{-}rand}}}$. That is, $y_1 = \mathsf{SPRE.Encode}(M_{i^*+1}, T, x_1, \mathsf{hk}; r_1)$.
   (c) It computes $\widetilde{M_x} \leftarrow \mathsf{Ckt.Obf}(\text{Prog-}(i^*, 1)[i^*, h, M, x_1, K_{\mathsf{crs}}, K'_{\mathrm{sim}}, K'_{\mathsf{SPRE}}, \mathsf{crs}[i^*+1], y_1, \mathsf{hk}], 1^\lambda)$ and sends $(\mathsf{crs}, \widetilde{M_x})$ to the adversary.
7. The adversary sends its guess b.

Hybrid H_4 In this hybrid, the challenger replaces the hardwired randomized encoding with a simulated one.

1. The adversary sends $M, x = (x_1, x_2), 1^o, T$ to the challenger.
2. The challenger computes $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$.
3. Then, the challenger chooses a key K_{crs} for the puncturable PRF F_1 and keys $K_{\mathrm{sim}}, K_{\mathsf{SPRE}}$ for the puncturable PRF F_2.

4. Next, the challenger computes the crs bits using K_{crs} for each $i \leq i^*$. It sets $\mathsf{crs}[i] = M(x)_i \oplus F_1(K_{\mathsf{crs}}, i)$ for each $i \in [i^*]$. For all $i > i^*$, the challenger chooses $\mathsf{crs}[i] \leftarrow \{0,1\}$.

5. The challenger then chooses hash function $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^* + 1)$. It computes $h = H(\mathsf{crs})$.

6. Finally, the challenger computes an obfuscation of $\mathsf{Prog}\text{-}(i^*, 1)$. It performs the following steps.

   (a) It first computes punctured keys $K'_{\mathsf{SPRE}} = K_{\mathsf{SPRE}}\{i^* + 1\} \leftarrow \mathsf{PPRF.Puncture}_2(K_{\mathsf{SPRE}}, i^* + 1)$, $K'_{\mathrm{sim}} = K_{\mathrm{sim}}\{i^* + 1\} \leftarrow \mathsf{PPRF.Puncture}_2(K_{\mathrm{sim}}, i^* + 1)$.

   (b) Next, it computes a simulated encoding y_1 of $M(x)_{i^*+1}$ with randomness $r_1 \leftarrow \{0,1\}^{\ell_{\text{sim-rand}}}$. That is, let $M(x)_{i^*+1}$ denote the $(i^* + 1)^{th}$ bit of $M(x)$. $y_1 = \mathsf{SPRE.Sim}(1^{|M_i|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, M(x)_{i^*+1}, T; r_1)$.

   (c) It computes $\widetilde{M_x} \leftarrow \mathsf{Ckt.Obf}(\mathsf{Prog}\text{-}(i^*, 1)[i^*, h, M, x_1, K_{\mathsf{crs}}, K'_{\mathrm{sim}}, K'_{\mathsf{SPRE}}, \mathsf{crs}[i^* + 1], y_1, \mathsf{hk}], 1^\lambda)$ and sends $(\mathsf{crs}, \widetilde{M_x})$ to the adversary.

7. The adversary sends its guess b.

Hybrid H_5 In this sub-hybrid, the challenger punctures the PRF key K_{crs} on index $i^* + 1$. It outputs an obfuscation of $\mathsf{Prog}\text{-}(i^*, 2)$.

1. The adversary sends $M, x = (x_1, x_2), 1^o, T$ to the challenger.

2. The challenger computes $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$.

3. Then, the challenger chooses a key K_{crs} for the puncturable PRF F_1 and keys $K_{\mathrm{sim}}, K_{\mathsf{SPRE}}$ for the puncturable PRF F_2.

4. Next, the challenger computes the crs bits using K_{crs} for each $i \leq i^*$. It sets $\mathsf{crs}[i] = M(x)_i \oplus F_1(K_{\mathsf{crs}}, i)$ for each $i \in [i^*]$. For all $i > i^*$, the challenger chooses $\mathsf{crs}[i] \leftarrow \{0,1\}^\lambda$.

5. The challenger then chooses hash function $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^* + 1)$. It computes $h = H(\mathsf{crs})$.

6. Finally, the challenger computes an obfuscation of $\mathsf{Prog}\text{-}(i^*, 2)$ defined in Figure 7. It performs the following steps.

   (a) It first computes punctured keys $K'_{\mathsf{SPRE}} = K_{\mathsf{SPRE}}\{i^* + 1\} \leftarrow \mathsf{PPRF.Puncture}_2(K_{\mathsf{SPRE}}, i^* + 1)$, $K'_{\mathrm{sim}} = K_{\mathrm{sim}}\{i^* + 1\} \leftarrow \mathsf{PPRF.Puncture}_2(K_{\mathrm{sim}}, i^* + 1)$.

   (b) Next, it computes a punctured PRF key $K'_{\mathsf{crs}} = K_{\mathsf{crs}}\{i^* + 1\} \leftarrow \mathsf{PPRF.Puncture}_1(K_{\mathsf{crs}}, i^* + 1)$.

   (c) Next, it computes a simulated encoding y_1 of $M(x)_{i^*+1}$ with randomness $r_1 \leftarrow \{0,1\}^{\ell_{\text{sim-rand}}}$. That is, let $M(x)_{i^*+1}$ denote the $(i^* + 1)^{th}$ bit of $M(x)$. $y_1 = \mathsf{SPRE.Sim}(1^{|M_i|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, M(x)_{i^*+1}, T; r_1)$.

   (d) It computes $\widetilde{M_x} \leftarrow \mathsf{Ckt.Obf}(\mathsf{Prog}\text{-}(i^*, 2)[i^*, h, M, x_1, K'_{\mathsf{crs}}, K'_{\mathrm{sim}}, K'_{\mathsf{SPRE}}, \mathsf{crs}[i^* + 1], y_1, \mathsf{hk}], 1^\lambda)$ and sends $(\mathsf{crs}, \widetilde{M_x})$ to the adversary.

7. The adversary sends its guess b.

$\mathsf{Prog}\text{-}(i^*, 2)$

Inputs : Index $i \in [o]$, bit $\mathsf{str} \in \{0,1\}$, proof π, index $j \in [t]$
Hardwired : Index i^*, Hash $h \in \{0,1\}^\lambda$, machine M, input x_1, punctured PRF keys K'_{crs}, $K'_{\mathrm{sim}}, K'_{\mathsf{SPRE}}$, bit $\mathsf{crs}[i^*+1]$, output y_1, preprocessed value hk, bound T.

1. Verify proof π : Check if $\mathsf{SSB.Verify}(H, h, i, \mathsf{str}, \pi) = 1$. If not, output $\perp$.
2. If $i \leq i^*$, do the following:
   (a) Let $w = F_1(K'_{\mathsf{crs}}, i)$ and y be the first bit of $w \oplus \mathsf{str}$.
   (b) Compute out $= \mathsf{SPRE.Sim}(1^{|M_i|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, y, T; r)$ where $r = F_2(K'_{\mathrm{sim}}, i)$.
   (c) Output j^{th} bit of out.
3. If $i = i^* + 1$ and $\mathsf{str} = \mathsf{crs}[i^*+1]$ then output j^{th} bit of y_1.
4. Else, if $i > i^*$: Recall that M_i denotes the turing machine that, on input x, runs the machine M on input x and outputs the i^{th} bit of $M(x)$. Compute out $= \mathsf{SPRE.Encode}(M_i, T, x_1, \mathsf{hk}; F_2(K'_{\mathsf{SPRE}}, i))$ and output the j^{th} bit of out.

Figure 7: Program $\mathsf{Prog}\text{-}(i^*, 2)$ used in sub-hybrid H_5

Hybrid H_6 In this sub-hybrid, the challenger replaces $\mathsf{crs}[i^*+1]$ with $M(x)_{i^*+1} \oplus F_1(K_{\mathsf{crs}}, i^*+1)$.

1. The adversary sends $M, x = (x_1, x_2), 1^o, T$ to the challenger.
2. The challenger computes $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$.
3. Then, the challenger chooses a key K_{crs} for the puncturable PRF F_1 and keys $K_{\mathrm{sim}}, K_{\mathsf{SPRE}}$ for the puncturable PRF F_2.
4. Next, the challenger computes the crs bits using K_{crs} for each $i \leq i^*$. It sets $\mathsf{crs}[i] = M(x)_i \oplus F_1(K_{\mathsf{crs}}, i)$ for each $i \in [i^*]$. For all $i > i^* + 1$, the challenger chooses $\mathsf{crs}[i] \leftarrow \{0,1\}$.
   It sets $\mathsf{crs}[i^*+1] = (M(x)_{i^*+1} \oplus F_1(K_{\mathsf{crs}}, i^*+1)$.
5. The challenger then chooses hash function $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^*+1)$. It computes $h = H(\mathsf{crs})$.
6. Finally, the challenger computes an obfuscation of $\mathsf{Prog}\text{-}(i^*, 2)$. It performs the following steps.
   (a) It first computes punctured keys $K'_{\mathsf{SPRE}} = K_{\mathsf{SPRE}}\{i^*+1\} \leftarrow \mathsf{PPRF.Puncture}_2(K_{\mathsf{SPRE}}, i^*+1)$, $K'_{\mathrm{sim}} = K_{\mathrm{sim}}\{i^*+1\} \leftarrow \mathsf{PPRF.Puncture}_2(K_{\mathrm{sim}}, i^*+1)$.
   (b) Next, it computes a punctured PRF key $K'_{\mathsf{crs}} = K_{\mathsf{crs}}\{i^*+1\} \leftarrow \mathsf{PPRF.Puncture}_1(K_{\mathsf{crs}}, i^*+1)$.
   (c) Next, it computes a simulated encoding y_1 of $M(x)_{i^*+1}$ with randomness $r_1 \leftarrow \{0,1\}^{\ell_{\text{sim-rand}}}$. That is, let $M(x)_{i^*+1}$ denote the $(i^*+1)^{th}$ bit of $M(x)$. $y_1 = \mathsf{SPRE.Sim}(1^{|M_i|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, M(x)_{i^*+1}, T; r_1)$.
   (d) It computes $\widetilde{M_x} \leftarrow \mathsf{Ckt.Obf}(\mathsf{Prog}\text{-}(i^*, 2)[i^*, h, M, x_1, K'_{\mathsf{crs}}, K'_{\mathrm{sim}}, K'_{\mathsf{SPRE}}, \mathsf{crs}[i^*+1], y_1, \mathsf{hk}], 1^\lambda)$ and sends $(\mathsf{crs}, \widetilde{M_x})$ to the adversary.
7. The adversary sends its guess b.

Hybrid H_7 In this sub-hybrid, the challenger computes the simulated encoding using randomness generated pseudorandomly with PRF key K_{sim}.

1. The adversary sends $M, x = (x_1, x_2), 1^o, T$ to the challenger.
2. The challenger computes $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$.

3. Then, the challenger chooses a key K_{crs} for the puncturable PRF F_1 and keys $K_{\text{sim}}, K_{\mathsf{SPRE}}$ for the puncturable PRF F_2.

4. Next, the challenger computes the crs bits using K_{crs} for each $i \le (i^*+1)$. It sets $\mathsf{crs}[i] = M(x)_i \oplus F_1(K_{\mathsf{crs}}, i)$ for each $i \in [i^*+1]$. For all $i > i^*+1$, the challenger chooses $\mathsf{crs}[i] \leftarrow \{0,1\}$.

5. The challenger then chooses hash function $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^*+1)$. It computes $h = H(\mathsf{crs})$.

6. Finally, the challenger computes an obfuscation of $\mathsf{Prog}\text{-}(i^*, 2)$. It performs the following steps.

   (a) It first computes punctured keys $K'_{\mathsf{SPRE}} = K_{\mathsf{SPRE}}\{i^*+1\} \leftarrow \mathsf{PPRF.Puncture}_2(K_{\mathsf{SPRE}}, i^*+1)$, $K'_{\text{sim}} = K_{\text{sim}}\{i^*+1\} \leftarrow \mathsf{PPRF.Puncture}_2(K_{\text{sim}}, i^*+1)$.

   (b) Next, it computes a punctured PRF key $K'_{\mathsf{crs}} = K_{\mathsf{crs}}\{i^*+1\} \leftarrow \mathsf{PPRF.Puncture}_1(K_{\mathsf{crs}}, i^*+1)$.

   (c) It computes a simulated encoding y_1 of $M(x)_{i^*+1}$ with randomness $r_1 = F_2(K_{\text{sim}}, i^*+1)$. That is, let $M(x)_{i^*+1}$ denote the $(i^*+1)^{th}$ bit of $M(x)$. $y_1 = \mathsf{SPRE.Sim}(1^{|M_i|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, M(x)_{i^*+1}, T; r_1)$.

   (d) It computes $\widetilde{M_x} \leftarrow \mathsf{Ckt.Obf}(\mathsf{Prog}\text{-}(i^*, 2)[i^*, h, M, x_1, K'_{\mathsf{crs}}, K'_{\text{sim}}, K'_{\mathsf{SPRE}}, \mathsf{crs}[i^*+1], y_1, \mathsf{hk}], 1^\lambda)$ and sends $(\mathsf{crs}, \widetilde{M_x})$ to the adversary.

7. The adversary sends its guess b.

Hybrid H_8 In this sub-hybrid, the challenger replaces all the punctured PRF keys $K'_{\mathsf{crs}}, K'_{\text{sim}}$ and K'_{SPRE} with unpunctured PRF keys.

1. The adversary sends $M, x = (x_1, x_2), 1^o, T$ to the challenger.

2. The challenger computes $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$.

3. Then, the challenger chooses a key K_{crs} for the puncturable PRF F_1 and keys $K_{\text{sim}}, K_{\mathsf{SPRE}}$ for the puncturable PRF F_2.

4. Next, the challenger computes the crs bits using K_{crs} for each $i \le (i^*+1)$. It sets $\mathsf{crs}[i] = M(x)_i \oplus F_1(K_{\mathsf{crs}}, i)$ for each $i \in [i^*+1]$. For all $i > i^*+1$, the challenger chooses $\mathsf{crs}[i] \leftarrow \{0,1\}$.

5. The challenger then chooses hash function $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^*+1)$. It computes $h = H(\mathsf{crs})$.

6. Finally, the challenger computes an obfuscation of $\mathsf{Prog}\text{-}(i^*, 3)$ defined in Figure 8. It performs the following steps.

   (a) Next, it computes a simulated encoding y_1 of $M(x)_{i^*+1}$ with randomness $r_1 = F_2(K_{\text{sim}}, i^*+1)$. That is, let $M(x)_{i^*+1}$ denote the $(i^*+1)^{th}$ bit of $M(x)$. $y_1 = \mathsf{SPRE.Sim}(1^{|M_i|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, M(x)_{i^*+1}, T; r_1)$.

   (b) It computes $\widetilde{M_x} \leftarrow \mathsf{Ckt.Obf}(\mathsf{Prog}\text{-}(i^*, 3)[i^*, h, M, x_2, K_{\mathsf{crs}}, K_{\text{sim}}, K_{\mathsf{SPRE}}, \mathsf{crs}[i^*+1], y_1, \mathsf{hk}], 1^\lambda)$ and sends $(\mathsf{crs}, \widetilde{M_x})$ to the adversary.

7. The adversary sends its guess b.

Hybrid H_9 In this sub-hybrid, the challenger changes the program being obfuscated to $\mathsf{Prog}\text{-}(i^*+1)$. This corresponds to Hyb_{i^*+1}.

1. The adversary sends $M, x = (x_1, x_2), 1^o, T$ to the challenger.

2. The challenger computes $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$.

3. Then, the challenger chooses a key K_{crs} for the puncturable PRF F_1 and keys $K_{\text{sim}}, K_{\mathsf{SPRE}}$ for the puncturable PRF F_2.

Prog-$(i^*, 3)$

Inputs : Index $i \in [o]$, bit $\mathsf{str} \in \{0,1\}$, proof π, index $j \in [t]$
Hardwired : Index i^*, Hash $h \in \{0,1\}^\lambda$, machine M, input x_1, PRF keys K_{crs}, K_{sim}, K_{SPRE}, bit $\mathsf{crs}[i^*+1]$, output y_1, preprocessed value hk, bound T.

1. Verify proof π : Check if $\mathsf{SSB.Verify}(H, h, i, \mathsf{str}, \pi) = 1$. If not, output $\perp$.
2. If $i \leq i^*$, do the following:
   (a) Let $w = F_1(K_{\mathsf{crs}}, i)$ and y be the first bit of $w \oplus \mathsf{str}$.
   (b) Compute out $= \mathsf{SPRE.Sim}(1^{|M_i|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, y, T; r)$ where $r = F_2(K_{\mathrm{sim}}, i)$.
   (c) Output j^{th} bit of out.
3. If $i = i^* + 1$ and $\mathsf{str} = \mathsf{crs}[i^*+1]$ then output j^{th} bit of y_1.
4. Else, if $i > i^*$: Recall that M_i denotes the turing machine that, on input (x, i), runs the machine M on input x and outputs the i^{th} bit of $M(x)$. Compute out $= \mathsf{SPRE.Encode}(M_i, T, x_1, \mathsf{hk}; F_2(K_{\mathsf{SPRE}}, i))$ and output the j^{th} bit of out.

Figure 8: Program Prog-$(i^*, 3)$ used in sub-hybrid H_7

4. Next, the challenger computes the crs bits using K_{crs} for each $i \leq (i^*+1)$. It sets $\mathsf{crs}[i] = M(x)_i \oplus F_1(K_{\mathsf{crs}}, i)$ for each $i \in [i^*+1]$. For all $i > i^*+1$, the challenger chooses $\mathsf{crs}[i] \leftarrow \{0,1\}$.
5. The challenger then chooses hash function $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^*+1)$. It computes $h = H(\mathsf{crs})$.
6. Finally, the challenger computes $\widetilde{M_x} \leftarrow \mathsf{Ckt.Obf}(\text{Prog-}(i^*+1), 1^\lambda)$ where Prog-(i^*+1) is defined in Figure 9. The challenger sends $(\mathsf{crs}, \widetilde{M_x})$ to the adversary.
7. The adversary sends its guess b.

Prog-(i^*+1)

Inputs : Index $i \in [o]$, bit $\mathsf{str} \in \{0,1\}$, proof π, index $j \in [t]$
Hardwired : Index (i^*+1), Hash $h \in \{0,1\}^\lambda$, machine M, input x_1, PRF keys $K_{\mathsf{crs}}, K_{\mathrm{sim}}, K_{\mathsf{SPRE}}$, preprocessed value hk, bound T.

1. Verify proof π : Check if $\mathsf{SSB.Verify}(H, h, i, \mathsf{str}, \pi) = 1$. If not, output $\perp$.
2. If $i \leq (i^*+1)$, do the following:
   (a) Let $w = F_1(K_{\mathsf{crs}}, i)$ and y be the first bit of $w \oplus \mathsf{str}$.
   (b) Compute out $= \mathsf{SPRE.Sim}(1^{|M_i|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, y, T; r)$ where $r = F_2(K_{\mathrm{sim}}, i)$.
   (c) Output j^{th} bit of out.
3. Else, if $i > (i^*+1)$: Recall that M_i denotes the turing machine that, on input (x, i), runs the machine M on input x and outputs the i^{th} bit of $M(x)$. Compute out $= \mathsf{SPRE.Encode}(M_i, T, x_1, \mathsf{hk}; F_2(K_{\mathsf{SPRE}}, i))$ and output the j^{th} bit of out.

Figure 9: Hybrid Program Prog-(i^*+1)

Indistinguishability of Sub-Hybrids:
We will now show that every pair of consecutive sub-hybrids is computationally indistinguishable and this completes the proof of Theorem 5.2.

Lemma 5.1. Assuming the index hiding property of the SSB hash system, hybrid H_0 is computationally indistinguishable from hybrid H_1.

Proof. Suppose there exists an adversary $\mathcal{A}$ that can distinguish between the two hybrids with non-negligible probability. We will use $\mathcal{A}$ to construct an adversary $\mathcal{A}_{\mathsf{SSB}}$ that breaks the index hiding property of the SSB hash scheme which is a contradiction.

The adversary $\mathcal{A}_{\mathsf{SSB}}$ sends the tuple (o, i^*, i^*+1) to the challenger $\mathcal{C}_{\mathsf{SSB}}$ of the SSB hash scheme. $\mathcal{C}_{\mathsf{SSB}}$ responds back with a hash key H that is binding either at index i^* or at index (i^*+1)- that is, responds either with $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^*)$ or $H \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^*+1)$.

Then, $\mathcal{A}_{\mathsf{SSB}}$ interacts with the adversary $\mathcal{A}$ and performs the experiment exactly as in Hybrid H_0 except that it sets the hash key H as the value received from $\mathcal{C}_{\mathsf{SSB}}$. Notice that when the challenger $\mathcal{C}_{\mathsf{SSB}}$ sends a hash key that is binding at index i^*, the experiment between $\mathcal{A}_{\mathsf{SSB}}$ and $\mathcal{A}$ corresponds exactly to Hybrid H_0 and when the challenger $\mathcal{C}_{\mathsf{SSB}}$ sends a hash key that is binding at index (i^*+1), the experiment between $\mathcal{A}_{\mathsf{SSB}}$ and $\mathcal{A}$ corresponds exactly to Hybrid H_1. Thus, if $\mathcal{A}$ can distinguish between the two hybrids with non-negligible probability, $\mathcal{A}_{\mathsf{SSB}}$ can use the same guess to break the index hiding property of the SSB hash scheme with non-negligible probability which is a contradiction. ∎

Lemma 5.2. Assuming the functionality preserved under puncturing property of the puncturable PRF F_2, the somewhere statistically binding property of the SSB hash system and the security of iO for circuits, hybrid H_1 is computationally indistinguishable from hybrid H_2.

Proof. Suppose there exists an adversary $\mathcal{A}$ that can distinguish between the two hybrids with non-negligible probability. We will use $\mathcal{A}$ to construct an adversary $\mathcal{A}_{i\mathcal{O}}$ that breaks the security of the indistinguishability obfuscator which is a contradiction.

The adversary $\mathcal{A}_{i\mathcal{O}}$ interacts with $\mathcal{A}$ exactly as in hybrid H_1 and receives the pair $(M, x, 1^o, T)$ from $\mathcal{A}$ in step 1. Then, $\mathcal{A}_{i\mathcal{O}}$ sends the pair of programs $(\mathsf{Prog}\text{-}i^*, \mathsf{Prog}\text{-}(i^*, 1))$ defined in Figure 5 and Figure 6 respectively to the challenger $\mathcal{C}_{i\mathcal{O}}$ of the indistinguishability obfuscation scheme. $\mathcal{C}_{i\mathcal{O}}$ picks one of the two programs randomly and responds back with an obfuscation of that program. Then, $\mathcal{A}_{i\mathcal{O}}$ sets this received value as $\widetilde{M_x}$ and performs the rest of the experiment with $\mathcal{A}$ exactly as in Hybrid H_1. Observe that when $\mathcal{C}_{i\mathcal{O}}$ picks program $\mathsf{Prog}\text{-}i^*$, the experiment between $\mathcal{A}_{i\mathcal{O}}$ and $\mathcal{A}$ corresponds exactly to Hybrid H_1 and when $\mathcal{C}_{i\mathcal{O}}$ picks program $\mathsf{Prog}\text{-}(i^*, 1)$, the experiment between $\mathcal{A}_{i\mathcal{O}}$ and $\mathcal{A}$ corresponds exactly to Hybrid H_2. From the security of the indistinguishability obfuscator, we know that if two programs are functionally equivalent, then their obfuscations are computationally indistinguishable. Thus, suppose the two programs $(\mathsf{Prog}\text{-}i^*, \mathsf{Prog}\text{-}(i^*, 1))$ were functionally equivalent, then, if $\mathcal{A}$ can distinguish between the two hybrids with non-negligible probability, $\mathcal{A}_{i\mathcal{O}}$ can use the same guess to break the security of the indistinguishability obfuscator with non-negligible probability which would be a contradiction.
We will now show that the two programs $(\mathsf{Prog}\text{-}i^*, \mathsf{Prog}\text{-}(i^*, 1))$ are functionally equivalent and this completes the proof. We will consider 3 cases that partition the set of inputs to the two programs.
Case 1: Input index $i \le i^*$
Now, by the functionality preserved under puncturing property of the puncturable PRF F_2, observe that $F_2(K_{\text{sim}}, i) = F_2(K'_{\text{sim}}, i)$ where K'_{sim} is a key punctured at index (i^*+1). Thus, both programs $\mathsf{Prog}\text{-}i^*$ and $\mathsf{Prog}\text{-}(i^*, 1)$ are functionally equivalent in this case.
Case 2: Input index $i > (i^*+1)$
Once again, by the functionality preserved under puncturing property of the puncturable PRF F_2, observe that $F_2(K_{\text{sim}}, i) = F_2(K'_{\text{sim}}, i)$. Thus, both programs $\mathsf{Prog}\text{-}i^*$ and $\mathsf{Prog}\text{-}(i^*, 1)$ are functionally equivalent in this case too.
Case 3: Input index $i = (i^*+1)$
Suppose the next part of the input - str equals $\mathsf{crs}[i^*+1]$. Then, observe that both programs output exactly the same value.

The only difference between the two programs' behavior is when the input is of the form $(i^*+1, \mathsf{str}, \pi, j)$ such that the proof π verifies and $\mathsf{str} \neq \mathsf{crs}[i^*+1]$. For such inputs, $\mathsf{Prog}\text{-}i^*$ runs step 3 - that is, computes $\text{out} = \mathsf{SPRE.Sim}(1^{|M_i|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, y, T; F_2(K_{\text{sim}}, i))$ and outputs the j^{th} bit whereas $\mathsf{Prog}\text{-}(i^*, 1)$ outputs $\perp$ since it can't evaluate $F_2(K'_{\mathsf{SPRE}}, i^*+1)$ in step 4. However, we will now show that for all inputs of the form $(i^*+1, \mathsf{str}, \pi, j)$ where $\mathsf{str} \neq \mathsf{crs}[i^*+1]$, there doesn't exist any value of π such that proof in step 1 verifies (i.e $\mathsf{SSB.Verify}(H, h, i^*+1, \mathsf{str}, \pi) = 1$) and hence both programs output the same value - $\perp$ on such inputs and this completes the proof. Observe that this is exactly the property guaranteed by the somewhere statistically binding property of the SSB hash system! Recall that, the somewhere statistically binding property states that, for $h \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^*+1)$, there doesn't exist $\mathsf{str}, \mathsf{str}'$ such that $\mathsf{str} \neq \mathsf{str}'$, π, π' such that $\mathsf{SSB.Verify}(H, h, i^*+1, \mathsf{str}, \pi) = \mathsf{SSB.Verify}(H, h, i^*+1, \mathsf{str}', \pi') = 1$. We know that since $h = H(\mathsf{crs})$, for $\mathsf{str} = \mathsf{crs}[i^*+1]$, there does exist π such that $\mathsf{SSB.Verify}(H, h, i^*+1, \mathsf{str}, \pi) = 1$. Thus, for all inputs of the form $(i^*+1, \mathsf{str}, \pi, j)$ where $\mathsf{str} \neq \mathsf{crs}[i^*+1]$, there doesn't exist any value of π such that proof in step 1 verifies and this completes the proof. ■

Lemma 5.3. Assuming the pseudorandom at punctured points property of the puncturable PRF F_2, hybrid H_2 is computationally indistinguishable from hybrid H_3.

Proof. Suppose there exists an adversary $\mathcal{A}$ that can distinguish between the two hybrids with non-negligible probability. We will use $\mathcal{A}$ to construct an adversary $\mathcal{A}_{\mathrm{PRF}}$ that breaks the pseudorandom at punctured points property of the puncturable PRF scheme which is a contradiction.

$\mathcal{A}_{\mathrm{PRF}}$ interacts with a challenger $\mathcal{C}_{\mathrm{PRF}}$ for the PRF scheme and sends a query with the point (i^*+1). $\mathcal{C}_{\mathrm{PRF}}$ picks a key K_{SPRE} for the puncturable PRF F_2 and responds back with a punctured key K'_{SPRE} that is computed as $K'_{\mathsf{SPRE}} \leftarrow \mathsf{PPRF.Puncture}_2(K_{\mathsf{SPRE}}, i^*+1)$. Then, $\mathcal{C}_{\mathrm{PRF}}$ also sends a value r^* which is picked either uniformly at random or is computed as $r^* = F_2(K_{\mathsf{SPRE}}, i^*+1)$. $\mathcal{A}_{\mathrm{PRF}}$ interacts with the adversary $\mathcal{A}$ exactly as in hybrid H_2 except that the randomness used to compute the encoding y_1 is set to be the value r^*. Notice that when the challenger $\mathcal{C}_{\mathrm{PRF}}$ computes r^* by evaluating the PRF, the experiment between $\mathcal{A}_{\mathrm{PRF}}$ and $\mathcal{A}$ corresponds exactly to Hybrid H_2 and when the challenger $\mathcal{C}_{\mathrm{PRF}}$ sends a uniformly random string, the experiment between $\mathcal{A}_{\mathrm{PRF}}$ and $\mathcal{A}$ corresponds exactly to Hybrid H_3. Thus, if $\mathcal{A}$ can distinguish between the two hybrids with non-negligible probability, $\mathcal{A}_{\mathrm{PRF}}$ can use the same guess to break the pseudorandom at punctured points property of the puncturable PRF F_2 with non-negligible probability which is a contradiction. ■

Lemma 5.4. Assuming the security of the succinct randomized encoding scheme SPRE, hybrid H_3 is computationally indistinguishable from hybrid H_4.

Proof. Suppose there exists an adversary $\mathcal{A}$ that can distinguish between the two hybrids with non-negligible probability. We will use $\mathcal{A}$ to construct an adversary $\mathcal{A}_{\mathsf{SPRE}}$ that breaks the security of the succinct randomized encoding scheme SPRE which is a contradiction.

The adversary $\mathcal{A}_{\mathsf{SPRE}}$ interacts with $\mathcal{A}$ exactly as in hybrid H_3 and receives the tuple $(M, x = (x_1, x_2), 1^o, T)$ from $\mathcal{A}$ in step 1. Then, $\mathcal{A}_{\mathsf{SPRE}}$ sends the tuple $(M_{i^*+1}, (x_1, x_2), T, 1^o)$ to the challenger $\mathcal{C}_{\mathsf{SPRE}}$ of the succinct randomized encoding scheme. $\mathcal{C}_{\mathsf{SPRE}}$ responds back with either an honestly generated encoding of (M_{i^*+1}, x) or a simulated one - that is, responds either with $\mathsf{SPRE.Encode}(M_{i^*+1}, T, x_1, \mathsf{hk}; r)$ or $\mathsf{SPRE.Sim}(1^{|M_{i^*+1}|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, M(x)_{i^*+1}, T; r)$ where $M(x)_{i^*+1}$ denotes the $(i^*+1)^{th}$ bit of $M(x)$, r is picked randomly and $\mathsf{hk} = \mathsf{SPRE.Preprocess}(1^\lambda, x_2)$. Then, $\mathcal{A}_{\mathsf{SPRE}}$ sets this received value as y_1 and performs the rest of the experiment with $\mathcal{A}$ exactly as in Hybrid H_3. Notice that when the challenger $\mathcal{C}_{\mathsf{SPRE}}$ sends an honest randomized encoding, the experiment between $\mathcal{A}_{\mathsf{SPRE}}$ and $\mathcal{A}$ corresponds exactly to Hybrid H_3 and when the challenger $\mathcal{C}_{\mathsf{SPRE}}$ sends a simulated randomized encoding, the experiment between $\mathcal{A}_{\mathsf{SPRE}}$ and $\mathcal{A}$ corresponds exactly to Hybrid H_4. Thus, if $\mathcal{A}$ can distinguish between the two hybrids with non-negligible probability, $\mathcal{A}_{\mathsf{SPRE}}$ can use the same guess to break the security of the succinct randomized encoding scheme SPRE with non-negligible probability which is a contradiction. ■

Lemma 5.5. Assuming the functionality preserved under puncturing property of the puncturable PRF F_1 and the security of iO for circuits, hybrid H_4 is computationally indistinguishable from hybrid H_5.

Proof. This proof is similar to the proof of Lemma 5.2. ■

Lemma 5.6. Assuming the pseudorandom at punctured points property of the puncturable PRF F_1, hybrid H_5 is computationally indistinguishable from hybrid H_6.

Proof. This proof is similar to the proof of Lemma 5.3. ∎

Lemma 5.7. Assuming the pseudorandom at punctured points property of the puncturable PRF F_2, hybrid H_6 is computationally indistinguishable from hybrid H_7.

Proof. This proof is similar to the proof of Lemma 5.3. ∎

Lemma 5.8. Assuming the functionality preserved under puncturing property of the puncturable PRFs F_1 and F_2 and the security of iO for circuits, hybrid H_7 is computationally indistinguishable from hybrid H_8.

Proof. This proof is similar to the proof of Lemma 5.2. ∎

Lemma 5.9. Assuming the somewhere statistically binding property of the SSB hash system and the security of iO for circuits, hybrid H_8 is computationally indistinguishable from hybrid H_9.

Proof. Suppose there exists an adversary $\mathcal{A}$ that can distinguish between the two hybrids with non-negligible probability. We will use $\mathcal{A}$ to construct an adversary $\mathcal{A}_{i\mathcal{O}}$ that breaks the security of the indistinguishability obfuscator which is a contradiction.

The adversary $\mathcal{A}_{i\mathcal{O}}$ interacts with $\mathcal{A}$ exactly as in hybrid H_8 and receives the pair $(M, x, 1^o, T)$ from $\mathcal{A}$ in step 1. Then, $\mathcal{A}_{i\mathcal{O}}$ sends the pair of programs $(\mathsf{Prog}\text{-}(i^*, 3), \mathsf{Prog}\text{-}(i^*+1))$ defined in Figure 8 and Figure 9 respectively to the challenger $\mathcal{C}_{i\mathcal{O}}$ of the indistinguishability obfuscation scheme. $\mathcal{C}_{i\mathcal{O}}$ picks one of the two programs randomly and responds back with an obfuscation of that program. Then, $\mathcal{A}_{i\mathcal{O}}$ sets this received value as $\widetilde{M_x}$ and performs the rest of the experiment with $\mathcal{A}$ exactly as in Hybrid H_8. Observe that when $\mathcal{C}_{i\mathcal{O}}$ picks program $\mathsf{Prog}\text{-}(i^*, 3)$, the experiment between $\mathcal{A}_{i\mathcal{O}}$ and $\mathcal{A}$ corresponds exactly to Hybrid H_8 and when $\mathcal{C}_{i\mathcal{O}}$ picks program $\mathsf{Prog}\text{-}(i^*+1)$, the experiment between $\mathcal{A}_{i\mathcal{O}}$ and $\mathcal{A}$ corresponds exactly to Hybrid H_9. From the security of the indistinguishability obfuscator, we know that if two programs are functionally equivalent, then their obfuscations are computationally indistinguishable. Thus, suppose the two programs $(\mathsf{Prog}\text{-}(i^*, 3), \mathsf{Prog}\text{-}(i^*+1))$ were functionally equivalent, then, if $\mathcal{A}$ can distinguish between the two hybrids with non-negligible probability, $\mathcal{A}_{i\mathcal{O}}$ can use the same guess to break the security of the indistinguishability obfuscator with non-negligible probability which would be a contradiction.
We will now show that the two programs $(\mathsf{Prog}\text{-}(i^*, 3), \mathsf{Prog}\text{-}(i^*+1))$ are functionally equivalent and this completes the proof. We will consider 2 cases that partition the set of inputs to the two programs.
Case 1: Input index $i \neq (i^*+1)$
It is easy to see that both programs $\mathsf{Prog}\text{-}i^*$ and $\mathsf{Prog}\text{-}(i^*, 1)$ are functionally equivalent in this case.
Case 2: Input index $i = (i^*+1)$
Suppose the next part of the input - str equals $\mathsf{crs}[i^*+1]$. Then, observe that both programs output exactly the same value.

The only difference between the two programs' behavior is when the input is of the form $(i^*+1, \mathsf{str}, \pi, j)$ such that the proof π verifies and $\mathsf{str} \neq \mathsf{crs}[i^*+1]$. For such inputs, $\mathsf{Prog}\text{-}(i^*+1)$ runs step 2 - that is, computes out $= \mathsf{SPRE.Sim}(1^{|M_i|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, y, T; F_2(K_{\mathrm{sim}}, i))$ and outputs the j^{th} bit whereas $\mathsf{Prog}\text{-}(i^*, 3)$ runs step 4. However, we will now show that for all inputs of the form $(i^*+1, \mathsf{str}, \pi, j)$ where $\mathsf{str} \neq \mathsf{crs}[i^*+1]$, there doesn't exist any value of π such that proof in step 1 verifies (i.e $\mathsf{SSB.Verify}(H, h, i^*+1, \mathsf{str}, \pi) = 1$) and hence both programs in fact output the same value - $\perp$ on such inputs and this completes the proof. Observe that this is exactly the property guaranteed by the somewhere statistically binding property of the SSB hash system! Recall that, the somewhere statistically binding property states that, for $h \leftarrow \mathsf{SSB.Gen}(1^\lambda, o, i^*+1)$, there doesn't exist $\mathsf{str}, \mathsf{str}'$ such that $\mathsf{str} \neq \mathsf{str}'$, π, π' such that $\mathsf{SSB.Verify}(H, h, i^*+1, \mathsf{str}, \pi) = \mathsf{SSB.Verify}(H, h, i^*+1, \mathsf{str}', \pi') = 1$. We know that since $h = H(\mathsf{crs})$, for $\mathsf{str} = \mathsf{crs}[i^*+1]$, there does exist π such that $\mathsf{SSB.Verify}(H, h, i^*+1, \mathsf{str}, \pi) = 1$. Thus, for all inputs of the form $(i^*+1, \mathsf{str}, \pi, j)$ where $\mathsf{str} \neq \mathsf{crs}[i^*+1]$, there doesn't exist any value of π such that proof in step 1 verifies and this completes the proof. ∎

■

6 Compact MPC

We consider the problem of constructing a malicious secure compact MPC protocol for Turing machines. Consider a set of n mutually distrusting parties with inputs $x_1, \dots, x_n$ respectively that agree on a turing machine M. Their goal is to securely compute the output $M(x_1, \dots, x_n)$ without leaking any information about their respective inputs where the output can be of any unbounded polynomial size. We first define the notion of a *compact* MPC protocol. Let λ denote the security parameter and let $\mathsf{Comm.Compl}(\pi)$ denote the communication complexity (sum of the lengths of all messages exchanged by all parties) of any protocol π. Let $\mathsf{Time}(\mathcal{M}, \mathsf{x})$ denote the running time of turing machine $\mathcal{M}$ on input x.

Definition 6.1. An MPC protocol π is said to be *compact* if there exists a fixed polynomial poly such that for all machines M and inputs $(x_1, \dots, x_n)$, $\mathsf{Comm.Compl}(\pi) = \mathsf{poly}(|M|, |x_1|, \dots, |x_n|, \lambda, \log(\mathsf{Time}(\mathcal{M}, \mathsf{x})))$. In particular, the communication complexity is independent of the output length and the running time of the machine on the inputs of all the parties.

In this section, we give a round preserving compiler from any constant round (non-compact) malicious secure MPC protocol in the plain model to a malicious secure *compact* MPC protocol for Turing machines in the random oracle (RO) model.

Formally, we prove the following theorem:

Theorem 6.1. For all $n, t > 0$, assuming the existence of:

- A (constant) k round [10] MPC protocol amongst n parties in the plain model that is malicious secure against up to t corruptions (AND)
- Strong output compressing randomized encodings in the shared randomness model,

there exists a k round *compact* MPC protocol π amongst n parties for Turing machines in the Programmable Random Oracle model that is malicious secure against up to t corruptions.

Here, we note that the above compiler even works if the underlying MPC protocol is for circuits. That is, we can convert any constant round protocol for circuits into a constant round protocol for Turing machines (with an input bound) by first converting the Turing machine into a (potentially large) circuit.

Corollaries:
We can instantiate the strong output compressing randomized encodings from our construction in Section 5. We now discuss several corollaries on instantiating the underlying MPC protocol with various protocols in literature based on different models.

1. Instantiating the MPC protocol with the round optimal[11] plain model construction of [BGJ+17a] that is secure against a dishonest majority based on DDH/Nth Residuosity, we get a four round *compact* MPC protocol π for Turing machines in the Programmable Random Oracle model that is malicious secure against a dishonest majority assuming iO for circuits and either DDH/Nth Residuosity.
2. We can also instantiate the underlying MPC protocol with protocols that are secure in the Common Random String model by using the RO's output on some fixed string to implement the common random string. In particular, combining the two round semi-malicious MPC protocol of [MW16] that is based on LWE in the common random string model or the ones of [GS18, BL18] that are based on DDH/N^{th}

[10]Observe that our round preserving compiler in fact works for any MPC protocol where the number of rounds is independent of the machine being evaluated.
[11]Recall that in the plain model, the optimal round complexity is 4.

residuosity in the plain model, with a non-interactive zero knowledge argument based on DLIN in the common random string model [GOS06], we get two round malicious secure MPC protocols in the common random string model. As a result, we have the following corollary:

Corollary 6.1. Assuming the existence of:

- iO for circuits (AND)
- A where A $\in \{LWE, DDH, N^{th}\ Residuosity\}$ (AND)
- DLIN

there exists a round optimal (two round) *compact* MPC protocol π for Turing machines in the Programmable Random Oracle model that is malicious secure against a dishonest majority.

3. We note that our transformation works even on instantiating the underlying constant round MPC protocol with ones that are secure in the setting of super-polynomial simulation[Pas03, BGI+17] or in the concurrent (self-composable) setting [GGJS12, BGJ+17b] to yield *compact* versions of the same in the RO model.

Implication to [HW15] Model. Finally, we observe that our transformation also has an implication to the circuit-based model of Hubácek and Wichs [HW15] as elaborated in Section 2.2. Thus, we get the following corollary:

Theorem 6.2. For all $n, t > 0$, assuming the existence of:

- A constant round MPC protocol amongst n parties in the plain model that is malicious secure against up to t corruptions (AND)
- Strong output compressing randomized encodings in the shared randomness model,

there exists a constant round MPC protocol π amongst n parties for all polynomial sized circuits in the Programmable Random Oracle model that is malicious secure against up to t corruptions where the communication complexity of the protocol is independent of the output length of the circuit. That is, there exists a fixed polynomial poly such that, for all circuits C and all inputs $(x_1, \ldots, x_n) \in \mathsf{Domain}(\mathsf{C})$, $\mathsf{Comm.Compl}(\pi) = \mathsf{poly}(|x_1|, \ldots, |x_n|, \lambda)$.

6.1 Construction

Notation and Primitives Used:

- Let λ denote the security parameter and RO be a random oracle that takes as input a tuple $(r, 1^{\mathsf{len}})$ where $|r| = \lambda$ and outputs a string of length len.
- Consider n parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ with inputs $\mathsf{x}_1, \ldots, \mathsf{x}_n$ respectively (with $|\mathsf{x}_i| = \lambda$ for each $i \in [n]$) who wish to evaluate any turing machine $\mathcal{M}$ on their joint inputs.
- Let $\mathsf{S.OcRE} = (\mathsf{S.OcRE.Setup}, \mathsf{S.OcRE.Encode}, \mathsf{S.OcRE.Decode})$ be a strong output compressing randomized encodings scheme in the shared randomness model.
- Let π^{plain} be a t round MPC protocol for turing machines in the plain model that is malicious secure against a dishonest majority. For simplicity, we assume that the protocol works using a broadcast channel - that is, in each round, every party broadcasts a message to all other parties.
- Let $\mathsf{NextMsg}_k(\cdot)$ denote the algorithm used by any party to compute the k^{th} round of protocol π^{plain} and let $\mathsf{Out}(\cdot)$ denote the algorithm used by any party to compute the final output. Also, without loss of generality, assume that in protocol π^{plain}, each party uses randomness rand_i of length λ. [12]

[12] Internally, we can apply a PRG to expand this to any length of randomness we require. Here, we are implicitly assuming that the protocol requires each party to use uniformly random strings. This is true of almost every constant round MPC protocol.

Remark: To ease the exposition, we assume that the Random Oracle can output arbitrarily long strings by also taking the desired output length as input to the oracle. In reality, let's say it outputs strings of length $p(\lambda)$ where p is a polynomial. Then, in the protocol below, each party can output a starting query index $r_{i,j}$ and an offset $o_{i,j}$ to indicate that the shared random string is actually the concatenation of $\mathsf{RO}(r_{i,j}), \ldots, \mathsf{RO}(r_{i,j} + o_{i,j})$. Note that $|o_{i,j}| \leq \lambda$.

Protocol: The protocol is described below.

1. **Round 1:**
   Each party P_i does the following:
   - Pick a random string $r_{i,1} \in \{0,1\}^\lambda$. Let $\mathsf{len}_{i,1} = |\mathsf{NextMsg}_1(\mathsf{x}_i; \mathrm{rand}_i)|$.
   - Compute $\mathsf{crs}_{i,1} = \mathsf{RO}(r_{i,1}, 1^{\mathsf{len}_{i,1}})$.
   - Compute $\mathsf{ek}_{i,1} = \mathsf{S.OcRE.Setup}(1^\lambda, 1^{\mathsf{len}_{i,1}}, \mathsf{crs}_{i,1})$.
   - Compute $\mathsf{msg}_{i,1} = \mathsf{S.OcRE.Encode}(\mathsf{NextMsg}_1, ((\mathsf{x}_i, \mathrm{rand}_i), \perp), 2^\lambda, \mathsf{ek}_{i,1})$.
   - Output $(\mathsf{msg}_{i,1}, r_{i,1}, \mathsf{len}_{i,1})$.
2. **Round 2 ... t:**
   For each subsequent round k, each party P_i does the following:
   - Let τ_{k-2} denote the transcript of the underlying protocol π^{plain} after round $(k-2)$. $\tau_0 = \perp$.
   - Set $\tau_{k-1} = \tau_{k-2}$.
   - For each party P_j, $(j \neq i)$ do the following:
     – Parse its previous round message as $(\mathsf{msg}_{j,k-1}, r_{j,k-1}, \mathsf{len}_{j,k-1})$.
     – Compute $\mathsf{crs}_{j,k-1} = \mathsf{RO}(r_{j,k-1}, 1^{\mathsf{len}_{j,k-1}})$.
     – Compute $\mathsf{msg}^{\mathsf{plain}}_{j,k-1} = \mathsf{S.OcRE.Decode}(\mathsf{msg}_{j,k-1}, \tau_{k-2}, \mathsf{crs}_{j,k-1})$.
     – Append $\mathsf{msg}^{\mathsf{plain}}_{j,k-1}$ to τ_{k-1}.
   - Pick a random string $r_{i,k} \in \{0,1\}^\lambda$. Let $\mathsf{len}_{i,k} = |\mathsf{NextMsg}_k(\mathsf{x}_i; \mathrm{rand}_i, \tau_{k-1})|$.
   - Compute $\mathsf{crs}_{i,k} = \mathsf{RO}(r_{i,k}, 1^{\mathsf{len}_{i,k}})$.
   - Compute $\mathsf{ek}_{i,k} = \mathsf{S.OcRE.Setup}(1^\lambda, 1^{\mathsf{len}_{i,k}}, \mathsf{crs}_{i,k})$.
   - Compute $\mathsf{msg}_{i,k} = \mathsf{S.OcRE.Encode}(\mathsf{NextMsg}_k, ((\mathsf{x}_i, \mathrm{rand}_i), \tau_{k-1}), 2^\lambda, \mathsf{ek}_{i,k})$.
   - Output[13] $(\mathsf{msg}_{i,k}, r_{i,k}, \mathsf{len}_{i,k})$.
3. **Output Computation:**
   Each party P_i does the following:
   - Let τ_{t-1} denote the transcript of the underlying protocol π^{plain} after round $(t-1)$.
   - Set $\tau_t = \tau_{t-1}$.
   - For each party P_j, $(j \neq i)$ do the following:
     – Parse its previous round message as $(\mathsf{msg}_{j,t}, r_{j,t}, \mathsf{len}_{j,t})$.
     – Compute $\mathsf{crs}_{j,t} = \mathsf{RO}(r_{j,t}, 1^{\mathsf{len}_{j,t}})$.
     – Compute $\mathsf{msg}^{\mathsf{plain}}_{j,t} = \mathsf{S.OcRE.Decode}(\mathsf{msg}_{j,t}, \tau_{t-1}, \mathsf{crs}_{j,t})$.
     – Append $\mathsf{msg}^{\mathsf{plain}}_{j,t}$ to τ_t.
   - Output $\mathsf{Out}(x_i, \mathrm{rand}_i, \tau_t)$.

[13]Note that to send $\mathsf{len}_{i,k}$, the length of the message is $\log \mathsf{len}_{i,k}$ and so at most λ.

Efficiency of the Protocol:
The size of the messages sent in round k by each party P_i is $3 \cdot \max\{|(\mathsf{msg}_{i,k}, r_{i,k}, \mathsf{len}_{i,k})|\}_{i,k}$. By the definition of strong output-compressing randomized encodings, $|\mathsf{msg}_{i,k}| \leq \mathsf{p}_2(|\mathsf{NextMsg}_k|, |(x_i, \mathrm{rand}_i)|, \log T, \lambda)$ where p_2 is a polynomial. $|\mathrm{rand}_i| = \lambda$, $|\mathsf{NextMsg}_k| = \mathsf{p}_3(|M|)$ where M is the original functionality and p_3 is a polynomial. Also, we know T is at most 2^λ. So $|\mathsf{msg}_{i,k}| \leq \mathsf{p}_3(|M|, |x_i|, \lambda)$ for some polynomial p_3. We know that $|r_{i,k}| = \lambda$ and $|\mathsf{len}_{i,k}| \leq \lambda$. Therefore, the size of the messages sent in round k by each party P_i is at most $\mathsf{p}_3(|M|, |x_i|, \lambda)$.

Since π^{plain} is a constant-round protocol, the total communication complexity of our protocol π is at most $\mathsf{p}(n, |M|, |x_1|, \ldots, |x_n|, \lambda)$ for a fixed polynomial p.

6.2 Security Proof

In this section, we formally prove Theorem 6.2.
Consider an adversary $\mathcal{A}$ who corrupts t parties where $t < n$. Let's say the simulator $\mathsf{Sim}^{\mathsf{plain}}$ for protocol π^{plain} consists of 4 algorithms $(\mathsf{Sim}_1^{\mathsf{plain}}, \mathsf{Sim}_2^{\mathsf{plain}}, \mathsf{Sim}_3^{\mathsf{plain}}, \mathsf{Sim}_{\mathsf{Out}}^{\mathsf{plain}})$ where: $\mathsf{Sim}_1^{\mathsf{plain}}(j, \cdot)$ outputs the adversary's view for the j^{th} of the first t_1 rounds, $\mathsf{Sim}_2^{\mathsf{plain}}$ queries the ideal functionality to receive the output, $\mathsf{Sim}_3^{\mathsf{plain}}(j, \cdot)$ outputs the adversary's view for the j^{th} round of the last $(t - t_1)$ rounds and $\mathsf{Sim}_{\mathsf{Out}}^{\mathsf{plain}}(i, \cdot)$ computes the output of honest party P_i. [14] Also, let's denote the size of $\mathsf{Sim}^{\mathsf{plain}}(\cdot)$ by $s(\lambda)$.

6.2.1 Description of Simulator

The strategy of the simulator Sim for our protocol π against a malicious adversary $\mathcal{A}$ is described below.

1. **Round 1 ...t_1:**
   For each round k and each honest party P_i, Sim does the following:
   - Let τ_{k-2} denote the transcript of the underlying protocol π^{plain} after round $(k-2)$. $\tau_0 = \perp$.
   - Set $\tau_{k-1} = \tau_{k-2}$.
   - For each party P_j, $(j \neq i)$, if $k > 1$, do the following:
     - Parse its previous round message as $(\mathsf{msg}_{j,k-1}, r_{j,k-1}, \mathsf{len}_{j,k-1})$.
     - Compute $\mathsf{crs}_{j,k-1} = \mathsf{RO}(r_{j,k-1}, 1^{\mathsf{len}_{j,k-1}})$.
     - Compute $\mathsf{msg}_{j,k-1}^{\mathsf{plain}} = \mathsf{S.OcRE.Decode}(\mathsf{msg}_{j,k-1}, \tau_{k-2}, \mathsf{crs}_{j,k-1})$.
     - Append $\mathsf{msg}_{j,k-1}^{\mathsf{plain}}$ to τ_{k-1}.
   - Compute $\mathsf{msg}_{i,k}^{\mathsf{plain}} = \mathsf{Sim}_1^{\mathsf{plain}}(k, \tau_{k-1}, \mathsf{st})$ where st denotes the state of $\mathsf{Sim}^{\mathsf{plain}}$.
   - Pick a random string $r_{i,k} \in \{0,1\}^\lambda$.
   - Compute $(\mathsf{msg}_{i,k}, \mathsf{crs}_{i,k}) \leftarrow \mathsf{S.OcRE.Sim}(1^{|s(\lambda|}, 1^{(2\cdot\lambda+|\tau_{k-1}|)}, 1^\lambda, \mathsf{msg}_{i,k}^{\mathsf{plain}}, 1^\lambda)$.
   - Set $\mathsf{RO}(r_{i,k}, 1^{|\mathsf{crs}_{i,k}|}) = \mathsf{crs}_{i,k}$.
   - Output[15] $(\mathsf{msg}_{i,k}, r_{i,k}, |\mathsf{crs}_{i,k}|)$.
2. **Query to Ideal Functionality:**
   Sim queries $\mathsf{Sim}_2^{\mathsf{plain}}(\tau_{k_1}, \mathsf{st})$ and receives an output y in return.
3. **Round $(t_1 + 1)$ ... t:**
   For each round k and each honest party P_i, Sim does the following:

[14] $\mathsf{Sim}_1^{\mathsf{plain}}$ also outputs some state that is fed as input to the subsequent algorithms and similarly for $\mathsf{Sim}_2^{\mathsf{plain}}, \mathsf{Sim}_3^{\mathsf{plain}}$.
[15] As before, note that to send the message $|\mathsf{crs}_{i,k}|$, the length of the string is $\log |\mathsf{crs}_{i,k}|$.

- Let τ_{k-2} denote the transcript of the underlying protocol π^{plain} after round $(k-2)$. $\tau_0 = \bot$.
- Set $\tau_{k-1} = \tau_{k-2}$.
- For each party P_j, $(j \neq i)$, if $k > 1$, do the following:
  - Parse its previous round message as $(\mathsf{msg}_{j,k-1}, r_{j,k-1}, \mathsf{len}_{j,k-1})$.
  - Compute $\mathsf{crs}_{j,k-1} = \mathsf{RO}(r_{j,k-1}, 1^{\mathsf{len}_{j,k-1}})$.
  - Compute $\mathsf{msg}^{\mathsf{plain}}_{j,k-1} = \mathsf{S.OcRE.Decode}(\mathsf{msg}_{j,k-1}, \tau_{k-2}, \mathsf{crs}_{j,k-1})$.
  - Append $\mathsf{msg}^{\mathsf{plain}}_{j,k-1}$ to τ_{k-1}.
- Compute $\mathsf{msg}^{\mathsf{plain}}_{i,k} = \mathsf{Sim}^{\mathsf{plain}}_3(k, y, \tau_{k-1}, \mathsf{st})$ where st denotes the state of $\mathsf{Sim}^{\mathsf{plain}}$.
- Pick a random string $r_{i,k} \in \{0,1\}^\lambda$.
- Compute $(\mathsf{msg}_{i,k}, \mathsf{crs}_{i,k}) \leftarrow \mathsf{S.OcRE.Sim}(1^{|s(\lambda|}, 1^{(2\cdot\lambda+|\tau_{k-1}|)}, 1^\lambda, \mathsf{msg}^{\mathsf{plain}}_{i,k}, 1^\lambda)$.
- Set $\mathsf{RO}(r_{i,k}, 1^{|\mathsf{crs}_{i,k}|}) = \mathsf{crs}_{i,k}$.
- Output $(\mathsf{msg}_{i,k}, r_{i,k}, |\mathsf{crs}_{i,k}|)$.

4. **Output Computation:**
   Sim does the following:

   - For each honest party P_i, do:
     - Let τ_{t-1} denote the transcript of the underlying protocol π^{plain} after round $(t-1)$.
     - Set $\tau_t = \tau_{t-1}$.
     - For each party P_j, $(j \neq i)$ do the following:
       * Parse its previous round message as $(\mathsf{msg}_{j,k-1}, r_{j,k-1}, \mathsf{len}_{j,k-1})$.
       * Compute $\mathsf{crs}_{j,k-1} = \mathsf{RO}(r_{j,k-1}, 1^{\mathsf{len}_{j,k-1}})$.
       * Compute $\mathsf{msg}^{\mathsf{plain}}_{j,t} = \mathsf{S.OcRE.Decode}(\mathsf{msg}_{j,t}, \tau_{t-1}, \mathsf{crs}_{j,t})$.
       * Append $\mathsf{msg}^{\mathsf{plain}}_{j,t}$ to τ_t.
     - If $\mathsf{Sim}^{\mathsf{plain}}_{\mathsf{Out}}(i, y, \tau_t, \mathsf{st}) = \bot$, send $\bot$ to the ideal functionality and stop.
   - Instruct the ideal functionality to deliver output to the honest parties.

Remarks: Note that if $\mathsf{Sim}^{\mathsf{plain}}$ is a rewinding simulator, our simulator Sim will also be a rewinding simulator.

6.2.2 Hybrids

We now show that the above simulation strategy is successful against all malicious PPT adversaries. That is, the view of the adversary along with the output of the honest parties is computationally indistinguishable in the real and ideal worlds. We will show this via a series of computationally indistinguishable hybrids where the first hybrid Hyb_0 corresponds to the real world and the last hybrid Hyb_2 corresponds to the ideal world.

1. Hyb_0 **- Real World:** In this hybrid, consider a simulator $\mathsf{Sim.Hyb}$ that plays the role of the honest parties.

2. Hyb_1 **- Simulate Encodings:** In this hybrid, in every round, $\mathsf{Sim.Hyb}$ computes the messages and the shared random string by running the simulator $\mathsf{S.OcRE.Sim}$ of the strong output compressing randomized encodings scheme. That is, for each round k and each honest party P_i, after decoding the transcript τ_{k-1} at the end of the previous round, $\mathsf{Sim.Hyb}$ does the following:

   - Compute $\mathsf{msg}^{\mathsf{plain}}_{i,k} = \mathsf{NextMsg}_k((\mathsf{x}_i, \mathrm{rand}_i), \tau_{k-1})$.
   - Pick a random string $r_{i,k} \in \{0,1\}^\lambda$.

- Compute $(\mathsf{msg}_{i,k}, \mathsf{crs}_{i,k}) \leftarrow \mathsf{S.OcRE.Sim}(1^{|\mathsf{NextMsg}_k|}, 1^{(2\cdot\lambda+|\tau_{k-1}|)}, 1^\lambda, \mathsf{msg}^{\mathsf{plain}}_{i,k}, 1^\lambda)$.
- Set $\mathsf{RO}(r_{i,k}, 1^{|\mathsf{crs}_{i,k}|}) = \mathsf{crs}_{i,k}$.
- Output $(\mathsf{msg}_{i,k}, r_{i,k}, |\mathsf{crs}_{i,k}|)$.

3. Hyb_2 **- Simulate MPC:** In this hybrid, in every round, $\mathsf{Sim.Hyb}$ computes the messages $\mathsf{msg}^{\mathsf{plain}}_{i,k}$ of the underlying protocol π^{plain} by running the simulator $\mathsf{Sim}^{\mathsf{plain}}$ of the strong output compressing randomized encodings scheme. The output computation phase is also performed exactly as done by Sim in the ideal world. This hybrid corresponds to the ideal world.

We will now show that every pair of successive hybrids is computationally indistinguishable.

Lemma 6.1. Assuming the security of the strong output compressing randomized encoding scheme $\mathsf{S.OcRE}$, Hyb_0 is computationally indistinguishable from Hyb_1.

Proof. Suppose there exists an adversary $\mathcal{A}$ that can distinguish between the two hybrids with non-negligible probability. We will use $\mathcal{A}$ to construct an adversary $\mathcal{A}_{\mathsf{S.OcRE}}$ that breaks the security of the scheme $\mathsf{S.OcRE}$ which is a contradiction.

$\mathcal{A}_{\mathsf{S.OcRE}}$ begins an execution of protocol π interacting with the adversary $\mathcal{A}$. Now, for each round k and every honest party P_i, let τ_{k-1} denote the transcript of the underlying protocol π^{plain}. $\mathcal{A}_{\mathsf{S.OcRE}}$ computes τ_{k-1} exactly as in Hyb_0. Then, $\mathcal{A}_{\mathsf{S.OcRE}}$ sends the tuple $(\mathsf{NextMsg}_k, ((x_i, \text{rand}_i), \tau_{k-1}), \lambda, T)$ to the challenger $\mathcal{C}_{\mathsf{OcRE}}$. Then, $\mathcal{C}_{\mathsf{S.OcRE}}$ sends back a pair of encoding and crs which is either honestly generated or simulated. $\mathcal{A}_{\mathsf{OcRE}}$ sets the received encoding as the value $\mathsf{msg}_{i,k}$ and the received crs as the value $\mathsf{crs}_{i,k}$. Then, $\mathcal{A}_{\mathsf{OcRE}}$ picks a random value $r_{i,k}$ of length λ. Since the adversary would have made only a polynomial number of queries so far to the random oracle out of a possible 2^λ choices for $r_{i,k}$, the probability that $r_{i,k}$ was queried to the random oracle before this is negligible. Then, $\mathcal{A}_{\mathsf{OcRE}}$ sets $\mathsf{RO}(r_{i,k}, 1^{|\mathsf{crs}_{i,k}|}) = \mathsf{crs}_{i,k}$. $\mathcal{A}_{\mathsf{S.OcRE}}$ sends $(\mathsf{msg}_{i,k}, r_{i,k}, |\mathsf{crs}_{i,k}|)$ to $\mathcal{A}$.

Notice that when the challenger $\mathcal{C}_{\mathsf{S.OcRE}}$ sends an honestly generated encoding and crs, the experiment between $\mathcal{A}_{\mathsf{S.OcRE}}$ and $\mathcal{A}$ corresponds exactly to Hyb_0 and when the challenger $\mathcal{C}_{\mathsf{S.OcRE}}$ sends a simulated encoding and crs, the experiment corresponds exactly to Hyb_1. Thus, if $\mathcal{A}$ can distinguish between the two hybrids with non-negligible probability, $\mathcal{A}_{\mathsf{S.OcRE}}$ can use the same guess to break the security of the scheme $\mathsf{S.OcRE}$ with non-negligible probability which is a contradiction. ∎

Lemma 6.2. Assuming the security of the MPC protocol plain, Hyb_1 is computationally indistinguishable from Hyb_2.

Proof. Suppose there exists an adversary $\mathcal{A}$ that can distinguish between the two hybrids with non-negligible probability. We will use $\mathcal{A}$ to construct an adversary $\mathcal{A}^{\mathsf{plain}}_\pi$ that breaks the security of the scheme π^{plain} which is a contradiction.

$\mathcal{A}^{\mathsf{plain}}_\pi$ begins an execution of protocol π for evaluating machine $\mathcal{M}$ interacting with the adversary $\mathcal{A}$ and an execution of protocol π^{plain} for evaluating machine $\mathcal{M}$ interacting with a challenger $\mathcal{C}^{\mathsf{plain}}_\pi$. Now, suppose $\mathcal{A}$ corrupts a set of parties $\mathcal{P}$, $\mathcal{A}_{\mathsf{plain}}$ corrupts the same set of parties in the protocol π^{plain}. For each round k and on behalf of every honest party P_i, $\mathcal{A}_{\mathsf{plain}}$ receives a message msg from the challenger $\mathcal{C}_{\mathsf{plain}}$. $\mathcal{A}_{\mathsf{plain}}$ sets msg to be the message $\mathsf{msg}_{i,k}$ in its interaction with $\mathcal{A}$ and then computes its messages to be sent to $\mathcal{A}$ exactly as in Hyb_1. Then, $\mathcal{A}^{\mathsf{plain}}_\pi$ decodes the messages sent by $\mathcal{A}$ in each round k and forwards them to $\mathcal{C}_{\mathsf{plain}}$ as its messages for protocol π^{plain} in round k. Finally, on behalf of the honest parties, $\mathcal{A}_{\mathsf{plain}}$ receives a set of outputs which is forwarded as the outputs of the honest parties in the protocol π to the adversary $\mathcal{A}$.

Notice that when the challenger $\mathcal{C}^{\mathsf{plain}}_\pi$ sends honestly generates messages, the experiment between $\mathcal{A}^{\mathsf{plain}}_\pi$ and $\mathcal{A}$ corresponds exactly to Hyb_1 and when the challenger $\mathcal{C}^{\mathsf{plain}}_\pi$ sends simulated messages, the experiment corresponds exactly to Hyb_2. Thus, if $\mathcal{A}$ can distinguish between the two hybrids with non-negligible probability, $\mathcal{A}^{\mathsf{plain}}_\pi$ can use the same guess to break the security of the scheme π^{plain} with non-negligible probability which is a contradiction. ∎

7 Constructing iO from Output-Compressing Randomized Encodings

In this section, we will show a construction of succinct iO for bounded-input turing machines from output-compressing randomized encodings (in the shared randomness model). Recall that we can construct output-compressing randomized encodings from strong output-compressing randomized encodings by just setting the public input x_2 to be $\perp$. [16] Also, recall from Section B.3 that in a succinct obfuscation scheme, the efficiency goal is to ensure that the size of the obfuscated turing machine is independent of the input bound. Formally, we show the following theorem:

Theorem 7.1. Assuming the existence of:

- Output-compressing randomized encodings in the shared randomness model,
- weakly succinct iO for bounded-input Turing machines,

There exists a succinct iO scheme for bounded-input Turing machines in the shared randomness model.

Instantiating the output-compressing randomized encodings from our construction in Section 5 and instantiating the weakly succinct obfuscation scheme with the scheme of [KLW15], we get the following corollary:

Corollary 7.1. Assuming the existence of:

- iO for circuits,
- A $\in$ {DDH, LWE, Nth Residuosity}.

There exists a succinct iO scheme for bounded-input Turing machines in the shared randomness model.

Notation and Primitives used:

- Let $\mathsf{OcRE} = (\mathsf{OcRE.Setup}, \mathsf{OcRE.Encode}, \mathsf{OcRE.Decode})$ be an output-compressing randomized encoding scheme (from Section 5.
- Let $(\mathsf{TM\text{-}bd.Obf}, \mathsf{TM\text{-}bd.Eval})$ be a weakly succinct obfuscation scheme for bounded-input Turing machines (where the size of obfuscation is allowed to depend on input bound). Such obfuscation schemes have been previously constructed in literature [KLW15]. Let the algorithm $\mathsf{TM\text{-}bd.Obf}$ be represented by a Turing machine $\mathsf{TM}_{\mathsf{TM\text{-}bd.Obf}}$ that takes as input a Turing machine description M, a time bound T, an input bound n (in unary), security parameter λ (in unary) and randomness r of length $\ell_{\mathrm{rnd}}(|M|, \lambda, n)$ (which is used to compute the obfuscated program).
- Let $\mathsf{Size}_{\mathsf{TM\text{-}bd.Obf}}(|M|, \lambda, n)$ denote the size of the obfuscation of a machine M using the scheme ($\mathsf{TM\text{-}bd.Obf}$, $\mathsf{TM\text{-}bd.Eval}$) with security parameter λ and input bound n (since $\mathsf{TM\text{-}bd.Obf}$ is an efficient algorithm, it follows that $\mathsf{Size}_{\mathsf{TM\text{-}bd.Obf}}$ is a polynomial in $(\lambda, n, |M|, \mathsf{log}(T))$.
- Let $\mathsf{TM}^*_{\mathsf{TM\text{-}bd.Obf},\mathrm{PRG}}$ be the Turing machine is described in Figure 10 and let $S_{\mathsf{TM\text{-}bd.Obf},\mathrm{PRG}}$ denote the size of the description of $\mathsf{TM}^*_{\mathsf{TM\text{-}bd.Obf},\mathrm{PRG}}$. (note that $S_{\mathsf{TM\text{-}bd.Obf},\mathrm{PRG}}$ is some constant that depends on the obfuscation scheme and the PRG scheme). Additionally,

[16] A similar transformation would also work in the standard model.

$\mathsf{TM}^*_{\mathsf{TM\text{-}bd.Obf},\mathrm{PRG}}$

Inputs : Machine M, time bound T, input bound 1^n, random string t.

1. Compute $r = \mathrm{PRG}(1^\lambda, t)$.
2. Output $\mathsf{TM}_{\mathsf{TM\text{-}bd.Obf}}(M, 1^n, 1^\lambda, r)$

Figure 10: Turing Machine $\mathsf{TM}^*_{\mathsf{TM\text{-}bd.Obf},\mathrm{PRG}}$

7.1 Construction

We will now describe a succinct iO scheme $\mathcal{O} = (\mathsf{TM.Setup}, \mathsf{TM.Obf}, \mathsf{TM.Eval})$ for bounded-input Turing machines in the shared randomness model, where the size of the obfuscation does not grow with the input bound.

$\mathsf{TM.Setup}(1^\lambda, 1^n, 1^m, \mathsf{crs} \in \{0,1\}^{\mathsf{Size}_{\mathsf{TM\text{-}bd.Obf}}(\lambda,n,m)})$: The setup algorithm does the following:

1. Set $o = \mathsf{Size}_{\mathsf{TM\text{-}bd.Obf}}(\lambda, n, m)$ and compute $\mathsf{ek} \leftarrow \mathsf{OcRE.Setup}(1^\lambda, 1^o, \mathsf{crs})$. Here m is used to denote the length of the largest turing machine in the class of turing machines we are interested to obfuscate.
2. Output $\mathsf{ok} = (\mathsf{ek}, n, \lambda)$ as the obfuscation key (both n and λ are represented in binary).

$\mathsf{TM.Obf}(M, T, \mathsf{ok})$: The obfuscation algorithm does the following:

1. Parse $\mathsf{ok} = (\mathsf{ek}, n, \lambda)$. Choose a uniformly random string $t \leftarrow \{0,1\}^\lambda$.
2. Set program $P \equiv \mathsf{TM}^*_{\mathsf{TM\text{-}bd.Obf},\mathrm{PRG}}$.
3. Output $\widetilde{M} \leftarrow \mathsf{OcRE.Encode}(P, M, n, \lambda, t, T, \mathsf{ek})$.

$\mathsf{TM.Eval}(\widetilde{M}, x, \mathsf{crs})$: The evaluation algorithm does the following:

1. Compute $M' \leftarrow \mathsf{OcRE.Decode}(\widetilde{M}, \mathsf{crs})$.
2. Output $\mathsf{TM\text{-}bd.Eval}(M', x)$.

Correctness: The correctness of the scheme follows easily by relying on the correctness of the underlying primitives used.

Efficiency:

- First, let us analyze the size of the obfuscation key. $|\mathsf{ok}| = |\mathsf{ek}| + \log(\lambda) + \log(n)$. Recall from the definition of OcRE that $\mathsf{ek} = p_1(\lambda, \log(o))$ where p_1 is a polynomial and $o = \mathsf{Poly}(\lambda, n, m)$. Thus, $|\mathsf{ok}| = p(\lambda, \log(n), \log(m))$ for some polynomial p.
- Next, let us analyze the size of the obfuscated program. The size of program P is $S_{\mathsf{TM\text{-}bd.Obf},\mathrm{PRG}}$ (which is a constant depending on the obfuscation and PRG scheme). The size of the input to P is $(|M| + \log(T) + \log(n) + \log(\lambda) + \lambda)$. Hence, the size of the encoding $\widetilde{M}$ (which is the final obfuscation of the turing machine M) is $p(|M|, \log(T), \lambda, \log(n), \log(m))$ for some polynomial p.
- Finally, let's analyze the running time of $\mathsf{TM.Eval}(\widetilde{M}, x, \mathsf{crs})$. We do this as follows.

  $\mathsf{Time}(\mathsf{TM.Eval}(\widetilde{M}, x, \mathsf{crs})) = \mathsf{Time}(\mathsf{OcRE.Decode}(\widetilde{M}, \mathsf{crs})) + \mathsf{Time}(\mathsf{TM\text{-}bd.Eval}(M', x))$. The running time of $\mathsf{OcRE.Decode}(\widetilde{M}, \mathsf{crs})$ is at most $\min(T, \mathsf{Time}(P, (M, n, \lambda, t, T))) \cdot p_3(\lambda, \log T = \lambda) \le p(|M|, n, \lambda)$. The running time of $\mathsf{TM\text{-}bd.Eval}(M', x)$ is at most $\min(T, \mathsf{Time}(M, x)) \cdot p(\lambda, n)$. So the running time $\mathsf{Time}(\mathsf{TM.Eval}(\widetilde{M}, x, \mathsf{crs})) \le p_1(|M|, n, \lambda) + \min(T, \mathsf{Time}(M, x)) \cdot p_2(\lambda, n) \le \min(T, \mathsf{Time}(M, x)) \cdot p_3(\lambda, n, |M|, \log T)$.

7.2 Security

The proof follows via a sequence of hybrid arguments as described below.

- Hyb_0**:** This is same as the original game. The challenger does the following:
    1. Receive a pair of functionally equivalent machines (M_0, M_1) and a time bound T from the adversary.
    2. Pick a bit b and a λ bit string t uniformly at random.
    3. Pick crs uniformly at random and compute $\mathsf{ok} = \mathsf{TM.Setup}(1^\lambda, 1^n, 1^m, \mathsf{crs})$.
    4. Set program $P \equiv \mathsf{TM}^*_{\mathsf{TM\text{-}bd.Obf},\mathrm{PRG}}$.
    5. Output $\widetilde{M} \leftarrow \mathsf{OcRE.Encode}(P, M_b, n, \lambda, t, T, \mathsf{ek})$ along with crs.
- Hyb_1**:** This is same as the original game. The challenger does the following:
    1. Receive a pair of functionally equivalent machines (M_0, M_1) and a time bound T from the adversary.
    2. Pick a bit b and a λ bit string t uniformly at random.
    3. Compute $(\mathsf{crs}, \widetilde{M}) \leftarrow \mathsf{OcRE.Sim}(1^{|\mathsf{TM\text{-}bd.Obf}|}, 1^{|(M_b, T, 1^\lambda, 1^n, \mathrm{PRG}(t))|}, 1^\lambda, \mathsf{TM\text{-}bd.Obf}(M_b, T, 1^\lambda, 1^n; \mathrm{PRG}(t)), 2^\lambda)$ and $\mathsf{ok} \leftarrow \mathsf{TM.Setup}(1^\lambda, 1^n, 1^m, \mathsf{crs})$.
    4. Output $(\mathsf{crs}, \mathsf{ek}, \widetilde{M})$.
- Hyb_2**:** This is same as the original game. The challenger does the following:
    1. Receive a pair of functionally equivalent machines (M_0, M_1) and a time bound T from the adversary.
    2. Pick a bit b uniformly at random.
    3. Compute $(\mathsf{crs}, \widetilde{M}) \leftarrow \mathsf{OcRE.Sim}(1^{|\mathsf{TM\text{-}bd.Obf}|}, 1^{|(M_b, T, 1^\lambda, 1^n, r_1)|}, 1^\lambda, \mathsf{TM\text{-}bd.Obf}(M_b, T, 1^\lambda, 1^n; r_1), 2^\lambda)$ where r_1 is picked uniformly at random and compute $\mathsf{ok} = \mathsf{TM.Setup}(1^\lambda, 1^n, 1^m, \mathsf{crs})$.
    4. Output $(\mathsf{crs}, \mathsf{ek}, \widetilde{M})$.

We will now argue that every pair of consecutive hybrids is computationally indistinguishable and finally, show that any PPT adversary has negligible advantage in Hyb_2.

Lemma 7.1. Assuming the security of the output compressing randomized encoding scheme OcRE, Hyb_0 is computationally indistinguishable from Hyb_1.

Proof. Suppose there exists an adversary $\mathcal{A}$ that can distinguish between the two hybrids with non-negligible probability. We will use $\mathcal{A}$ to construct an adversary $\mathcal{A}_{\mathsf{OcRE}}$ that breaks the security of the scheme OcRE which is a contradiction.

First, $\mathcal{A}_{\mathsf{OcRE}}$ receives a pair of functionally equivalent machines (M_0, M_1) and a time bound T from $\mathcal{A}$. Then, $\mathcal{A}_{\mathsf{OcRE}}$ picks a bit b uniformly at random, a λ bit string t uniformly at random, sets $P \equiv \mathsf{TM}^*_{\mathsf{TM\text{-}bd.Obf},\mathrm{PRG}}$ sends the tuple $(P, M_b, n, \lambda, t, T)$ to the challenger $\mathcal{C}_{\mathsf{OcRE}}$. Then, $\mathcal{C}_{\mathsf{OcRE}}$ sends back a pair of encoding and crs (shared randomness) which is either honestly generated or simulated. Then, $\mathcal{A}_{\mathsf{OcRE}}$ sends these values to $\mathcal{A}$ as its crs and the obfuscated output.

Notice that when the challenger $\mathcal{C}_{\mathsf{OcRE}}$ sends an honestly generated encoding and crs, the experiment between $\mathcal{A}_{\mathsf{OcRE}}$ and $\mathcal{A}$ corresponds exactly to Hyb_0 and when the challenger $\mathcal{C}_{\mathsf{OcRE}}$ sends a simulated encoding and crs, the experiment corresponds exactly to Hyb_1. Thus, if $\mathcal{A}$ can distinguish between the two hybrids with non-negligible probability, $\mathcal{A}_{\mathsf{OcRE}}$ can use the same guess to break the security of the scheme OcRE with non-negligible probability which is a contradiction. ■

Lemma 7.2. Assuming the security of the pseudorandom generator PRG, Hyb_1 is computationally indistinguishable from Hyb_2.

Proof. Suppose there exists an adversary $\mathcal{A}$ that can distinguish between the two hybrids with non-negligible probability. We will use $\mathcal{A}$ to construct an adversary $\mathcal{A}_{\text{PRG}}$ that breaks the security of the pseudorandom generator PRG which is a contradiction.

The adversary $\mathcal{A}_{\text{PRG}}$ interacts with a challenger $\mathcal{C}_{\text{PRG}}$ and receives a string which is either an output of the PRG or a uniformly random string. Then, $\mathcal{A}_{\text{PRG}}$ interacts with the adversary $\mathcal{A}$ and performs the experiment exactly as in Hyb_1 except that it sets the randomness r as the value received from $\mathcal{C}_{\text{PRG}}$. Notice that when the challenger $\mathcal{C}_{\text{PRG}}$ sends a PRG output, the experiment between $\mathcal{A}_{\text{PRG}}$ and $\mathcal{A}$ corresponds exactly to Hyb_1 and when the challenger $\mathcal{C}_{\mathsf{SSB}}$ sends a uniformly random string, the experiment corresponds exactly to Hyb_2. Thus, if $\mathcal{A}$ can distinguish between the two hybrids with non-negligible probability, $\mathcal{A}_{\text{PRG}}$ can use the same guess to break the security of the PRG with non-negligible probability which is a contradiction. ∎

Lemma 7.3. Assuming the security of the indistinguishability obfuscation scheme $(\mathsf{TM\text{-}bd.Obf}, \mathsf{TM\text{-}bd.Eval})$, the adversary's advantage in Hyb_2 is negligible.

Proof. Suppose there exists an adversary $\mathcal{A}$ that has a non-negligible advantage in distinguishing between $b = 0$ and $b = 1$ in Hyb_2. We will use $\mathcal{A}$ to construct an adversary $\mathcal{A}_i\mathcal{O}$ that breaks the security of the indistinguishability obfuscation scheme $(\mathsf{TM\text{-}bd.Obf}, \mathsf{TM\text{-}bd.Eval})$ which is a contradiction.

$\mathcal{A}_i\mathcal{O}$ receives a pair of functionally equivalent machines (M_0, M_1) and a time bound T from the adversary. Then, $\mathcal{A}_i\mathcal{O}$ sends the tuple $(M_0, M_1, T, 1^\lambda, 1^n)$ to the challenger $\mathcal{C}$ of the obfuscation scheme and receives back an obfuscation y of one of the two machines M_0 or M_1 using the algorithm $\mathsf{TM\text{-}bd.Obf}$. $\mathcal{A}$ uses this value to compute a simulated $\mathsf{crs} \leftarrow \mathsf{OcRE.Sim}(y; r)$, $\mathsf{ok} = \mathsf{TM.Setup}(1^\lambda, 1^n, 1^m, \mathsf{crs})$ and $\widetilde{M_x} \leftarrow \mathsf{OcRE.Sim}(y; r)$ using a random string r. $\mathcal{A}_{\mathsf{OcRE}}$ sends $(\mathsf{crs}, \widetilde{M_x})$ to the adversary $\mathcal{A}$ as its output.

Notice that when the challenger $\mathcal{C}_i\mathcal{O}$ sends an obfuscation of M_0, the experiment between $\mathcal{A}_i\mathcal{O}$ and $\mathcal{A}$ corresponds exactly to Hyb_2 using $b = 0$ and when the challenger $\mathcal{C}_i\mathcal{O}$ sends an obfuscation of M_1, the experiment between $\mathcal{A}_i\mathcal{O}$ and $\mathcal{A}$ corresponds exactly to Hyb_2 using $b = 1$. Thus, if $\mathcal{A}$ has a non-negligible advantage in distinguishing between $b = 0$ and $b = 1$ in Hyb_2, $\mathcal{A}_i\mathcal{O}$ can use the same distinguishing guess $\mathcal{A}$ to break the security of the indistinguishability obfuscation scheme $(\mathsf{TM\text{-}bd.Obf}, \mathsf{TM\text{-}bd.Eval})$ which is a contradiction. ∎

7.3 Succinct Partial Randomized Encodings

In this section, we introduce the notion of succinct partial randomized encodings (spRE). This is similar to the notion of succinct randomized encodings (defined in B.4), except that the adversary is allowed to learn part of the input. For efficiency, we require that if the machine has size m, and ℓ bits of input are hidden, then the size of randomized encoding should be polynomial in the security parameter λ, ℓ and m. In particular, the size of the encoding does not depend on the entire input's length (this is possible only because we want to hide ℓ bits of the input; the adversary can learn the remaining bits of the input). This notion is the Turing Machine analogue of *partial garbling* of arithmetic branching programs, studied by Ishai and Wee [IW14].

A succinct partial randomized encoding scheme SPRE for a class of boolean Turing machines $\mathcal{M}$ consists of a preprocessing algorithm $\mathsf{Preprocess}$, encoding algorithm Encode, and a decoding algorithm Decode with the following syntax.

$\mathsf{Preprocess}(1^\lambda, x_2 \in \{0,1\}^*)$: The preprocessing algorithm takes as input security parameter λ (in unary), string $y \in \{0,1\}^*$ and outputs a string hk.

$\mathsf{Encode}(M \in \mathcal{M}, T \in \mathbb{N}, x_1 \in \{0,1\}^*, \mathsf{hk} \in \{0,1\}^{p(\lambda)})$: The encoding algorithm takes as input a Turing machine $M \in \mathcal{M}$, time bound $T \in \mathbb{N}$, partial input $x_1 \in \{0,1\}^*$, string $\mathsf{hk} \in \{0,1\}^{p(\lambda)}$, and outputs an encoding $\widetilde{M}$.

$\mathsf{Decode}(\widetilde{M}, x_2, \mathsf{hk})$: The decoding algorithm takes as input an encoding $\widetilde{M}$, a string $x_2 \in \{0,1\}^*$, string hk and outputs $y \in \{0, 1, \perp\}$.

Definition 7.1. Let $\mathcal{M}$ be a family of Turing machines. A randomized encoding scheme $\mathsf{SPRE} = (\mathsf{Preprocess}, \mathsf{Encode}, \mathsf{Decode})$ is said to be a succinct partial randomized encoding scheme if it satisfies the following correctness, efficiency and security properties.

- Correctness: For every machine $M \in \mathcal{M}$, string $x = (x_1, x_2) \in \{0,1\}^*$, security parameter λ and $T \in \mathbb{N}$, if $\mathsf{hk} \leftarrow \mathsf{Preprocess}(1^\lambda, x_2)$, then $\mathsf{Decode}(\mathsf{Encode}(M, T, x_1, \mathsf{hk}), x_2) = \mathsf{TM}(M, x, T)$.
- Efficiency: There exist polynomials $p_{\mathsf{prep}}, p_{\mathrm{enc}}$ and p_{dec} such that for every machine $M \in \mathcal{M}$, $x = (x_1, x_2) \in \{0,1\}^*$, $T \in \mathbb{N}$ and $\lambda \in \mathbb{N}$, if $\mathsf{hk} \leftarrow \mathsf{Preprocess}(1^\lambda, x_2)$, then $|\mathsf{hk}| = p_{\mathsf{prep}}(\lambda)$, the time to encode $\widetilde{M} \leftarrow \mathsf{Encode}(M, T, x_1, \mathsf{hk})$ is bounded by $p_{\mathrm{enc}}(|M|, |x_1|, \log T, \lambda)$, and the time to decode $\widetilde{M}$ is bounded by $\min(\mathsf{Time}(M, x, T) \cdot p_{\mathrm{dec}}(\lambda, \log T)$.
- Security: For every PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a PPT simulator $\mathcal{S}$ such that for all PPT distinguishers $\mathcal{D}$, there exists a negligible function $\mathrm{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$, $\Pr[1 \leftarrow \mathcal{D}(\mathsf{Expt\text{-}SPRE\text{-}Real}_{\mathsf{SPRE},\mathcal{A}}(\lambda))] - \Pr[1 \leftarrow \mathcal{D}(\mathsf{Expt\text{-}SPRE\text{-}Ideal}_{\mathsf{SRE},\mathcal{A},\mathcal{S}}(\lambda))] \leq \mathrm{negl}(\lambda)$, where $\mathsf{Expt\text{-}SPRE\text{-}Real}$ and $\mathsf{Expt\text{-}SPRE\text{-}Ideal}$ are defined in Figure 11. Moreover, the running time of $\mathcal{S}$ is bounded by some polynomial $p_{\mathcal{S}}(|M|, |x_1|, \log T, \lambda)$.

Experiments $\mathsf{Expt\text{-}SPRE\text{-}Real}_{\mathsf{SPRE},\mathcal{A}}(\lambda)$ **and** $\mathsf{Expt\text{-}SPRE\text{-}Ideal}_{\mathsf{SPRE},\mathcal{A},\mathcal{S}}(\lambda)$

$\mathsf{Expt\text{-}SPRE\text{-}Real}_{\mathsf{SPRE},\mathcal{A}}(\lambda)$:

- $(M, x = (x_1, x_2), T, \sigma) \leftarrow \mathcal{A}_1(1^\lambda)$.
- $\mathsf{hk} \leftarrow \mathsf{Preprocess}(x_2, 1^\lambda)$.
- $\widetilde{M} \leftarrow \mathsf{Encode}(M, T, x_1, \mathsf{hk})$.
- Experiment outputs $\mathcal{A}_2(\widetilde{M}, \sigma)$.

$\mathsf{Expt\text{-}SPRE\text{-}Ideal}_{\mathsf{SPRE},\mathcal{A},\mathcal{S}}(\lambda)$:

- $(M, x = (x_1, x_2), T, \sigma) \leftarrow \mathcal{A}_1(1^\lambda)$, $t^* = \min(T, \mathsf{Time}(M, x))$, $\mathrm{out} = \mathsf{TM}(M, x, T)$.
- $\mathsf{hk} \leftarrow \mathsf{Preprocess}(1^\lambda, x_2)$.
- $\widetilde{M} \leftarrow \mathcal{S}\left(1^{|M|}, 1^{|x_1|}, \mathsf{hk}, 1^\lambda, \mathrm{out}, t^*\right)$.
- Experiment outputs $\mathcal{A}_2(\widetilde{M}, \sigma)$.

Figure 11: Simulation Security Experiments for partial randomized encodings

8 Construction of Succinct Partial Randomized Encodings

We will now present a succinct partial RE scheme. This construction is similar to the succinct RE (machine hiding) scheme by [KLW15].

Overview of construction Our construction is closely related to the SRE scheme by [KLW15] (referred to as 'KLW scheme' for the remaining section). Let us first recall the KLW scheme (at a high level). The KLW construction assumes the Turing machine is oblivious, has a single work tape, which initially has the input and the next-step function, at any step, reads a state and symbol, and outputs the new state, and the new symbol written at that position. In this scheme, the encoding of machine M and input x consists of a obfuscated circuit Prog (which corresponds to the encoding of M), an encryption of the input x, encryption of initial TM state, a hash of the initial work tape (which has encryption of the input), and a signature on this hash, inital state's encryption (together with some additional components). The circuit Prog takes as input a time step, an encrypted state, an encrypted symbol, a proof and a signature. It first verifies the proof and signature. Then it decrypts the state and symbol, computes the new state and symbol (using the next-step circuit of M), encrypts the new state and symbol, updates the hash, computes a new signature on the hash, encrypted state (and additional components) and outputs the new encrypted state, encrypted symbol, hash and signature. The decoder runs this program iteratively. Let us assume the machine reads position p at time step i. Also, at time step i, let $\mathsf{ct}_{\mathrm{sym}}$ be the encryption of symbol, $\mathsf{ct}_{\mathrm{st}}$ the ciphertext at position p on worktape, h the hash of encrypted tape and σ the signature. The decoder, at step i, first computes a proof π that ct is the correct ciphertext at position p. It then runs Prog on input $i, h, \mathsf{ct}_{\text{sym}}, \mathsf{ct}_{\text{st}}, \pi, \sigma$, and receives new ciphertexts $\mathsf{ct}'_{\text{sym}}, \mathsf{ct}'_{\text{st}}$, new hash h' and signature σ'. It replaces ct_{sym} with $\mathsf{ct}'_{\text{sym}}$ on worktape at position p, and this concludes the i^{th} iteration. This procedude is repeated till the program outputs $0/1$.

Our construction: We will assume that the Turing machine M has a separate 'auxiliary-input tape' and work tape, the Turing machine is oblivious. The input to the Turing machine is split into actual input and auxiliary input. The auxiliary input is written on the aux-tape, the work tape initially contains the remaining input. The next-step function takes as input the current worktape symbol, current aux-tape symbol, current state and outputs new worktape symbol and new state.

In the preprocessing stage, we choose hash function H_{aux}, and computes $h_{\mathsf{aux}} = H_{\mathsf{aux}}(x_2)$. The preprocessing stage outputs H_{aux} together with h_{aux}.

The encoding algorithm, on input $M, x_1, H_{\mathsf{aux}}, h_{\mathsf{aux}}$, first chooses another hash function H_{wk} which will be used to compute a hash of the encrypted worktape. It outputs an obfuscation of circuit Prog, encryption ct of x_1 (which is the initial worktape), hash $h = H_{\text{wk}}(\mathsf{ct})$, encryption ct_0 of initial state and signatue on $h_{\mathsf{prep}}, h, \mathsf{ct}_0$. The circuit Prog is similar to the KLW circuit. It takes as input a time step, an encrypted worktape symbol, an encrypted state, aux-input symbol, hash of input and work tape, two corresponding proofs and a signature. It first verifies the signature and the two proofs. Next, it decrypts and obtains the worktape symbol, the state and uses these two, together with the current aux-input symbol to compute the new worktape symbol and state. It then encrypts the worktape symbol and state, updates the worktape hash, and outputs a new signature. Decoding is identical to the KLW decoding.

Our Construction Our construction requires the following primitives:

- A secret key encryption scheme $\mathsf{SKE} = (\mathsf{SKE.Setup}, \mathsf{SKE.Enc}, \mathsf{SKE.Dec})$. We will assume $\mathsf{SKE.Setup}$ uses $\ell_1 = \ell_1(\lambda)$ bits of randomness, and $\mathsf{SKE.Enc}$ uses $\ell_2 = \ell_2(\lambda)$ bits of randomness, where ℓ_1 and ℓ_2 are polynomials and let $\ell_{\text{rnd}} = \ell_1 + 2\ell_2$. We will let ℓ_3 denote the bit length of ciphertexts produced by $\mathsf{SKE.Enc}$.
- A secure indistinguishability obfuscator scheme $(\mathsf{Ckt.Obf}, \mathsf{Ckt.Eval})$ for circuit family $\{\mathcal{C}_\lambda\}_{\lambda\in\mathbb{N}}$, where each circuit in $\mathcal{C}_\lambda$ has size $\mathsf{s}_{\mathsf{ckt}}(\lambda)$, depth $\mathsf{dep}_{\mathsf{ckt}}(\lambda)$, takes $\mathsf{inp}_{\mathsf{ckt}}(\lambda)$ bits as input, and outputs $\mathsf{out}_{\mathsf{ckt}}(\lambda)$ bits, for some polynomials $\mathsf{s}_{\mathsf{ckt}}, \mathsf{dep}_{\mathsf{ckt}}, \mathsf{inp}_{\mathsf{ckt}}, \mathsf{out}_{\mathsf{ckt}}$.
- A positional accumulator scheme Acc =($\mathsf{Acc.Setup}$, $\mathsf{Acc.Setup\text{-}Enforce\text{-}Read}$, $\mathsf{Acc.Setup\text{-}Enforce\text{-}Write}$, $\mathsf{Acc.Prep\text{-}Read}$, $\mathsf{Acc.Prep\text{-}Write}$, $\mathsf{Acc.Verify\text{-}Read}$, $\mathsf{Acc.Write\text{-}Store}$, Update) with message space $\{0,1\}^{\ell_3 + \lg T}$ and producing accumulator values of bit length ℓ_{Acc}.
- An iterator Itr = ($\mathsf{Itr.Setup}$, $\mathsf{Itr.Setup\text{-}Enforce}$, $\mathsf{Itr.Iterate}$) for message space $\{0,1\}^{\ell_3 + \ell_{\mathsf{Acc}} + \lg T}$ with iterated value of size ℓ_{Itr} bits
- A splittable signature scheme $\mathcal{S} = (\mathsf{Spl.Setup}, \mathsf{Spl.Sign}, \mathsf{Spl.Verify}, \mathsf{Spl.Split}, \mathsf{Spl.Sign\text{-}abo})$ with message space $\{0,1\}^{\ell_{\mathsf{Itr}} + \ell_3 + \ell_{\mathsf{Acc}} + \lg T}$. For simplicity of notation, we will assume $\mathsf{Spl.Setup}$ uses $\ell_{\text{rnd}}(\lambda)$ bits of randomness.
- A puncturable PRF $\mathsf{PPRF} = (F, F.\mathsf{Setup}, F.\mathsf{Puncture})$ with key space $\mathcal{K}$, punctured key space $\mathcal{K}_p$, domain $[T]$, range $\{0,1\}^{\ell_{\text{rnd}}(\lambda)}$.
- A somewhere statistically binding scheme $\mathsf{SSB} = (\mathsf{SSB.Gen}, \mathsf{SSB.Open}, \mathsf{SSB.Verify})$.

We will now define the preprocessing, encoding and decoding algorithms.

$\mathsf{Preprocess}(1^\lambda, x_2)$: The preprocessing algorithm does the following:

1. Choose $H_{\mathsf{aux}} \leftarrow \mathsf{SSB.Gen}(1^\lambda, 1^{|x_2|}, 1)$.
2. Compute $h_{\mathsf{aux}} = H_{\mathsf{aux}}(x_2)$ and output $(H_{\mathsf{aux}}, h_{\mathsf{aux}}, 1^\lambda)$.

$\mathsf{Encode}(M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}, \mathsf{tmf}_{\text{wk}}, \mathsf{tmf}_{\mathsf{aux}}\rangle, T, x_1, (H_{\mathsf{aux}}, h_{\mathsf{aux}}, 1^\lambda))$: The encoding algorithm does the following:

1. Choose puncturable PRF keys $K_E \leftarrow F.\mathsf{Setup}(1^\lambda)$, $K_A \leftarrow F.\mathsf{Setup}(1^\lambda)$. K_E will be used for computing an encryption of the symbol and state, and K_A to compute the secret key/verification key for signature scheme.
2. Let $(r_{0,1}, r_{0,2}, r_{0,3}) = F(K_E, 0)$, $\mathsf{sk} = \mathsf{SKE.Setup}(1^\lambda; r_{0,1})$.
3. Let $\ell_{\mathsf{inp}} = |x_1|$. Encrypt each bit of x_1 separately; that is, compute $\mathsf{ct}_i = \mathsf{SKE.Enc}(\mathsf{sk}, x_{1,i})$ for $1 \le i \le \ell_{\mathsf{inp}}$. These ciphertexts are 'accumulated' using the accumulator.
4. Compute $(\mathsf{P}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{\text{STORE}_0}) \leftarrow \mathsf{Acc.Setup}(1^\lambda, T)$. Then, for $1 \le j \le \ell_{\mathsf{inp}}$, compute $\widetilde{\text{STORE}}_j = \mathsf{Acc.Write\text{-}Store}(\mathsf{P}_{\mathsf{Acc}}, \widetilde{\text{STORE}}_{j-1}, j-1, (\mathsf{ct}_j, 0))$, $\mathsf{acc\text{-}aux}_j = \mathsf{Acc.Prep\text{-}Write}(\mathsf{P}_{\mathsf{Acc}}, \widetilde{\text{STORE}}_{j-1}, j-1)$, $\widetilde{w}_j = \mathsf{Update}(\mathsf{P}_{\mathsf{Acc}}, \widetilde{w}_{j-1}, \mathsf{inp}_j, j-1, \mathsf{acc\text{-}aux}_j)$.
5. Set $w_0 = \widetilde{w}_{\ell_{\mathsf{inp}}}$ and $s_0 = \widetilde{\text{STORE}}_{\ell_{\mathsf{inp}}}$.
6. Compute $(\mathsf{P}_{\mathsf{Itr}}, v_0) \leftarrow \mathsf{Itr.Setup}(1^\lambda, T)$.
7. Then, compute an obfuscation $P \leftarrow \mathsf{Ckt.Obf}(1^\lambda, \mathsf{Prog}\{M,\ T,\ \mathsf{P}_{\mathsf{Acc}},\ \mathsf{P}_{\mathsf{Itr}},\ K_E,\ K_A, H_{\mathsf{aux}}, h_{\mathsf{aux}}\})$ where Prog is defined in Figure 12.
8. Compute $\mathsf{ct}_{\mathsf{st}} \leftarrow \mathsf{SKE.Enc}(\mathsf{sk}, q_0)$.
9. Let $r_A = F(K_A, 0)$, $(\mathrm{SK}_0, \mathrm{VK}_0, \mathrm{VK}_{0,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$ and $\sigma_0 = \mathsf{Spl.Sign}(\mathrm{SK}_A, (v_0, \mathsf{ct}_{\mathsf{st}}, w_0, 0))$.
10. Output $\mathsf{Enc} = (P, w_0, v_0, \sigma_0, s_0)$.

$\mathsf{Decode}(\mathsf{Enc}, x_2, (H_{\mathsf{aux}}, h_{\mathsf{aux}}))$: The decoding algorithm receives as input $\mathsf{Enc} = ((P, \mathsf{ct}_{\mathsf{st},0}, w_0, v_0, \sigma_0, \text{STORE}_0))$. It first sets $\mathsf{pos}_0 = 0$ and $\mathsf{pos}_{\mathsf{aux},0} = 0$. Then, for $i = 1$ to T, it does the following:

1. Let $((\mathsf{ct}_{\mathsf{sym},\ell\text{-}\mathsf{w}}, \ell\text{-}\mathsf{w}), \pi) = \mathsf{Acc.Prep\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, \text{STORE}_{i-1}, \mathsf{pos}_{i-1})$.
2. Let $\mathsf{acc\text{-}aux} = \mathsf{Acc.Prep\text{-}Write}(\mathsf{P}_{\mathsf{Acc}}, \text{STORE}_{i-1}, \mathsf{pos}_{i-1})$.
3. Let $\mathsf{sym}_{\mathsf{aux}} = x_{2,\mathsf{tmf}_{\mathsf{aux}}(i-1)}$, $\pi_{\mathsf{aux}} = \mathsf{SSB.Open}(H_{\mathsf{aux}}, x_2, \mathsf{tmf}_{\mathsf{aux}}(i-1))$.
4. Compute $(\mathsf{pos}_i, (\mathsf{ct}_{\mathsf{sym},i}, \ell\text{-}\mathsf{w}), \mathsf{ct}_{\mathsf{st},i}, w_i, v_i, \sigma_i) = P(t, (\mathsf{ct}_{\mathsf{sym},\ell\text{-}\mathsf{w}}, \ell\text{-}\mathsf{w}), \mathsf{ct}_{\mathsf{st},i-1}, w_{i-1}, v_{i-1}, \sigma_{i-1}, \mathsf{acc\text{-}aux}, \pi, \pi_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}})$. If P has output 0, 1, or $\perp$, then output the same.
5. Otherwise, compute $\text{STORE}_i = \mathsf{Acc.Write\text{-}Store}(\mathsf{P}_{\mathsf{Acc}}, \text{STORE}_{i-1}, \mathsf{pos}_i, (\mathsf{ct}_{\mathsf{sym},i}, i))$.

8.1 Correctness and Efficiency

Program Prog

Constants: Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_{\mathrm{wk}}, \mathsf{tmf}_{\mathsf{aux}}\rangle$, time bound T, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A \in \mathcal{K}$, SSB Hash function H_{aux} and hash value h_{aux}.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value v_{in}, signature σ_{in}, accumulator proof π, SSB proof π_{aux}, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

```
1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. Verifications
    (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
    (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
    (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
    (d) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
3. Computing next state and symbol (encrypted)
    (a) Let $(r_{\ell\text{-w},1}, r_{\ell\text{-w},2}, r_{\ell\text{-w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
    (b) Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
    (c) Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
    (d) If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
    (e) Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
4. Update accumulator, iterator and compute new signature
    (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
    (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
    (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
    (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$ and $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
5. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```

Figure 12: Program Prog

8.2 Proof of Security

Our security proof is similar to the KLW security proof. The main differences are as follows:

- KLW has an indistinguishability-based security definition, while we require simulation security. However, it is easy to note that the KLW construction also satisfies simulation security.
- We have an additional auxiliary input, and we need to make sure this does not leak additional information about the machine or hidden input. To ensure this, we will keep the overall KLW proof structure, and make local modifications to the KLW proof. In particular, the only parts that change are the steps where KLW use the accumulator enforcing property. Together with the accumulator enforcement, we also need to make the SSB hash binding at appropriate position. If, at time t, positions p and p_{aux} are read, then we need to make the accumulator enforcing at position p, and the SSB hash binding at p_{aux}.

Simulator S: We will now describe the simulator S. The simulator takes as input $1^{|M|}, \mathsf{tmf}, \mathsf{tmf}_{\mathsf{aux}}$, $1^{|x|}$, $(H_{\mathsf{aux}}, h_{\mathsf{aux}})$, 1^{λ}, t^*, res, where $t^* = \min\left(T, \mathsf{Time}\left(M, (x, y)\right)\right)$ and $\mathsf{res} = \mathsf{TM}\left(M, (x, y), T\right)$ and does the following.

- It first chooses puncturable PRF keys $K_E \leftarrow F.\mathsf{Setup}(1^{\lambda})$, $K_A \leftarrow F.\mathsf{Setup}(1^{\lambda})$. Let $(r_{0,1}, r_{0,2}, r_{0,3}) = F(K_E, 0)$, $\mathsf{sk} = \mathsf{SKE.Setup}(1^{\lambda}; r_{0,1})$.
- It computes $(\mathsf{P}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{\mathrm{STORE}}_0) \leftarrow \mathsf{Acc.Setup}(1^{\lambda}, T)$. Let $\ell_{\mathsf{inp}} = |x|$, and let erase be a symbol not in Σ_{tape}. It encrypts $\mathsf{erase}^{\ell_{\mathsf{inp}}}$; that is, it computes $\mathsf{ct}_i = \mathsf{SKE.Enc}(\mathsf{sk}, \mathsf{erase})$ for $1 \leq i \leq \ell_{\mathsf{inp}}$. It also computes $\mathsf{ct}_{\mathsf{st}} \leftarrow \mathsf{SKE.Enc}(\mathsf{sk}, \mathsf{erase}^{\ell_{\mathrm{st}}})$ (where ℓ_{st} is the number of bits required to represent each state of Turing machine).
- Next, it computes $\mathsf{acc\text{-}aux}_j = \mathsf{Acc.Prep\text{-}Write}(\mathsf{P}_{\mathsf{Acc}}, \widetilde{\mathrm{STORE}}_{j-1}, j-1)$, $\widetilde{\mathrm{STORE}}_j = \mathsf{Acc.Write\text{-}Store}(\mathsf{P}_{\mathsf{Acc}}, \widetilde{\mathrm{STORE}}_{j-1}, j-1, (\mathsf{ct}_j, 0))$, $\widetilde{w}_j = \mathsf{Update}(\mathsf{P}_{\mathsf{Acc}}, \widetilde{w}_{j-1}, \mathsf{inp}_j, j-1, \mathsf{acc\text{-}aux}_j)$ for $1 \leq j \leq \ell_{\mathsf{inp}}$, and sets $w_0 = \widetilde{w}_{\ell_{\mathsf{inp}}}$, $s_0 = \widetilde{\mathrm{STORE}}_{\ell_{\mathsf{inp}}}$.
- It then computes $(\mathsf{P}_{\mathsf{Itr}}, v_0) \leftarrow \mathsf{Itr.Setup}(1^{\lambda}, T)$.
- Finally, it computes an obfuscation $P \leftarrow \mathsf{Ckt.Obf}(1^{\lambda}, \mathsf{Prog\text{-}sim}\{\mathsf{tmf}, \mathsf{tmf}_{\mathsf{aux}}, t^*, \mathsf{res}, \mathsf{P}_{\mathsf{Acc}}, \mathsf{P}_{\mathsf{Itr}}, K_E, K_A, H_{\mathsf{aux}}, h_{\mathsf{aux}}\})$ where $\mathsf{Prog\text{-}sim}$ is defined in Figure 13. This program is padded to be of the same size as Prog (defined in Figure 12).
- Let $r_A = F(K_A, 0)$, $(\mathrm{SK}_0, \mathrm{VK}_0) = \mathsf{Spl.Setup}(1^{\lambda}; r_A)$ and $\sigma_0 = \mathsf{Spl.Sign}(\mathrm{SK}_A, (v_0, \mathsf{ct}_{\mathsf{st}}, w_0, 0))$. It outputs $\mathsf{Enc} = (P, w_0, v_0, \sigma_0, s_0)$.

Theorem 8.1. Assuming SKE is $\mathsf{IND\text{-}CPA}$ secure, $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF, Itr is an iterator satisfying Definitions C.9 and C.10, Acc is an accumulator satisfying Definitions C.5, C.6, C.7 and C.8, $\mathcal{S}$ is a splittable signature scheme satisfying security Definitions C.1, C.2, C.3 and C.4, SSB is a somewhere statistically binding scheme satisfying Definition B.1, the above scheme is a secure succinct partial randomized encoding scheme satisfying Definition 7.1.

The security proof is included in Appendix D (we include the full proof for completeness, the majority of which is borrowed from [KLW15]).

Program $\mathsf{Prog\text{-}sim}$

Constants: Tape movement functions $\mathsf{tmf}, \mathsf{tmf}_{\mathsf{aux}}$, Time bound $t^* \in \mathbb{N}$, Output res, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A \in \mathcal{K}$, SSB Hash function H_{aux} and hash value h_{aux}.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value v_{in}, signature σ_{in}, accumulator proof π, SSB proof π_{aux}, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

```
1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}(t)$.
2. If $t > t^*$, output $\perp$.
3. Verifications
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
4. Simulated Output
   (a) If $t = t^*$, output $\mathsf{res}$.
   (b) Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}^{\ell_{\mathsf{st}}}; r_{t,3})$.
5. Update accumulator, iterator and compute new signature
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$ and $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
6. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```

Figure 13: Program $\mathsf{Prog\text{-}sim}$

References

[AGVW13] Shweta Agrawal, Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption: New perspectives and lower bounds. In Canetti and Garay [CG13], pages 500–518.

[AIKW13] Benny Applebaum, Yuval Ishai, Eyal Kushilevitz, and Brent Waters. Encoding functions with constant online rate or how to compress garbled circuits keys. In Canetti and Garay [CG13], pages 166–184.

[AJ15] Prabhanjan Ananth and Abhishek Jain. Indistinguishability obfuscation from compact functional encryption. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 308–326. Springer, 2015.

[AJS17] Prabhanjan Ananth, Abhishek Jain, and Amit Sahai. Indistinguishability obfuscation for turing machines: Constant overhead and amortization. In Katz and Shacham [KS17], pages 252–279.

[AL18] Prabhanjan Ananth and Alex Lombardi. Succinct garbling schemes from functional encryption through a local simulation paradigm. Cryptology ePrint Archive, Report 2018/759, 2018. `https://eprint.iacr.org/2018/759`.

[AM18] Shweta Agrawal and Monosij Maitra. Functional encryption and indistinguishability obfuscation for turing machines from minimal assumptions, 2018.

[BGI+17] Saikrishna Badrinarayanan, Sanjam Garg, Yuval Ishai, Amit Sahai, and Akshay Wadia. Two-message witness indistinguishability and secure computation in the plain model from new assumptions. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part III*, volume 10626 of *Lecture Notes in Computer Science*, pages 275–303. Springer, 2017.

[BGJ+17a] Saikrishna Badrinarayanan, Vipul Goyal, Abhishek Jain, Yael Kalai, Dakshita Khurana, and Amit Sahai. Promise zero knowledge and its applications to round optimal mpc. *IACR Cryptology ePrint Archive*, 2017:1088, 2017.

[BGJ+17b] Saikrishna Badrinarayanan, Vipul Goyal, Abhishek Jain, Dakshita Khurana, and Amit Sahai. Round optimal concurrent MPC via strong simulation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 743–775. Springer, 2017.

[BGL+15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang. Succinct randomized encodings and their applications. In Servedio and Rubinfeld [SR15], pages 439–448.

[BL18] Fabrice Benhamouda and Huijia Lin. k-round mpc from k-round ot via garbled interactive circuits. *EUROCRYPT*, 2018.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.*, pages 62–73, 1993.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In Yuval Ishai, editor, *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, volume 6597 of *Lecture Notes in Computer Science*, pages 253–273. Springer, 2011.

[CDG+18] Can Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. *EUROCRYPT*, 2018.

[CG13] Ran Canetti and Juan A. Garay, editors. *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*. Springer, 2013.

[CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Succinct garbling and indistinguishability obfuscation for RAM programs. In Servedio and Rubinfeld [SR15], pages 429–437.

[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 Canetti and Garay [CG13], pages 519–535.

[CJS14] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 597–608. ACM, 2014.

[DSW08] Yevgeniy Dodis, Victor Shoup, and Shabsi Walfish. Efficient constructions of composable commitments and zero-knowledge proofs. In David A. Wagner, editor, *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, volume 5157 of *Lecture Notes in Computer Science*, pages 515–535. Springer, 2008.

[GGJS12] Sanjam Garg, Vipul Goyal, Abhishek Jain, and Amit Sahai. Concurrently secure computation in constant rounds. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 99–116. Springer, 2012.

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

[GGMP16] Sanjam Garg, Divya Gupta, Peihan Miao, and Omkant Pandey. Secure multiparty RAM computation in constant rounds. In Hirt and Smith [HS16], pages 491–520.

[GHL+14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. 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 405–422. Springer, 2014.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[GOS06] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Non-interactive zaps and new techniques for NIZK. In Cynthia Dwork, editor, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*, volume 4117 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2006.

[GS18] Sanjam Garg and Akshayaram Srinivasan. A simple construction of io for turing machines. Cryptology ePrint Archive, Report 2018/771, 2018. `https://eprint.iacr.org/2018/771`.

[HJK$^+$16] Dennis Hofheinz, Tibor Jager, Dakshita Khurana, Amit Sahai, Brent Waters, and Mark Zhandry. How to generate and use universal samplers. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, volume 10032 of *Lecture Notes in Computer Science*, pages 715–744, 2016.

[HS16] Martin Hirt and Adam D. Smith, editors. *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I*, volume 9985 of *Lecture Notes in Computer Science*, 2016.

[HW15] Pavel Hubácek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. In Tim Roughgarden, editor, *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11-13, 2015*, pages 163–172. ACM, 2015.

[HY16] Carmit Hazay and Avishay Yanai. Constant-round maliciously secure two-party computation in the RAM model. In Hirt and Smith [HS16], pages 521–553.

[IK00] Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 294–304. IEEE Computer Society, 2000.

[IMS12] Yuval Ishai, Mohammad Mahmoody, and Amit Sahai. On efficient zero-knowledge pcps. In Ronald Cramer, editor, *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*, volume 7194 of *Lecture Notes in Computer Science*, pages 151–168. Springer, 2012.

[IW14] Yuval Ishai and Hoeteck Wee. Partial garbling schemes and their applications. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 650–662. Springer, 2014.

[KLW14] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. Cryptology ePrint Archive, Report 2014/925, 2014. `https://eprint.iacr.org/2014/925`.

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In Servedio and Rubinfeld [SR15], pages 419–428.

[KS17] Jonathan Katz and Hovav Shacham, editors. *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II*, volume 10402 of *Lecture Notes in Computer Science*. Springer, 2017.

[Lin17] Yehuda Lindell. How to simulate it–a tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer, 2017.

[LO17] Steve Lu and Rafail Ostrovsky. Black-box parallel garbled RAM. In Katz and Shacham [KS17], pages 66–92.

[LPST16] Huijia Lin, Rafael Pass, Karn Seth, and Sidharth Telang. Output-compressing randomized encodings and applications. In Eyal Kushilevitz and Tal Malkin, editors, *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part I*, volume 9562 of *Lecture Notes in Computer Science*, pages 96–124. Springer, 2016.

[Mia16] Peihan Miao. Cut-and-choose for garbled RAM. *IACR Cryptology ePrint Archive*, 2016:907, 2016.

[MW16] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 735–763. Springer, 2016.

[Nie02] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *Annual International Cryptology Conference*, pages 111–126. Springer, 2002.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In Frank Thomson Leighton and Peter W. Shor, editors, *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 294–303. ACM, 1997.

[Pas03] Rafael Pass. Simulation in quasi-polynomial time, and its application to protocol composition. In Eli Biham, editor, *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, volume 2656 of *Lecture Notes in Computer Science*, pages 160–176. Springer, 2003.

[SR15] Rocco A. Servedio and Ronitt Rubinfeld, editors. *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*. ACM, 2015.

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 475–484. ACM, 2014.

[Wee09] Hoeteck Wee. Zero knowledge in the random oracle model, revisited. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 417–434. Springer, 2009.

A Secure Multiparty Computation in the Random Oracle Model

Parts of this section have been taken from [Gol04], with modifications for the random oracle model using [Lin17, Wee09, Nie02] as references.

A multi-party protocol is cast by specifying a random process that maps pairs of inputs to pairs of outputs (one for each party). We refer to such a process as a functionality. The security of a protocol is defined with respect to a functionality f. In particular, let n denote the number of parties. A non-reactive n-party functionality f is a (possibly randomized) mapping of n inputs to n outputs. A multiparty protocol in the programmable random oracle model with security parameter λ for computing a non-reactive functionality f is a protocol running in time $\mathsf{poly}(\lambda)$ where there are parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$ with inputs $(x_1, \ldots, x_n)$ respectively, and additionally there is one other party RO which implements a random oracle functionality and which any party can query. It satisfies the following correctness requirement: if all run an honest execution of the protocol, then the joint distribution of the outputs $y_1, \ldots, y_n$ of the parties is statistically close to $f(x_1, \ldots, x_n)$.

A reactive functionality f is a sequence of non-reactive functionalities $f = (f_1, \ldots, f_\ell)$ computed in a stateful fashion in a series of phases. Let x_i^j denote the input of P_i in phase j, and let s^j denote the state of the computation after phase j. Computation of f proceeds by setting s^0 equal to the empty string and then computing $(y_1^j, \ldots, y_n^j, s^j) \leftarrow f_j(s^{j-1}, x_1^j, \ldots, x_n^j)$ for $j \in [\ell]$, where y_i^j denotes the output of P_i at the end of phase j. A multi-party protocol computing f also runs in ℓ phases, at the beginning of which each party holds an input and at the end of which each party obtains an output. (Note that parties may wait to decide on their phase-j input until the beginning of that phase.) Parties maintain state throughout the entire execution. The correctness requirement is that, in an honest execution of the protocol, the joint distribution of all the outputs $\{y_1^j, \ldots, y_n^j\}_{j=1}^{\ell}$ of all the phases is statistically close to the joint distribution of all the outputs of all the phases in a computation of f on the same inputs used by the parties.

Defining Security. The security of a protocol (with respect to a functionality f) is defined by comparing the real-world execution of the protocol with an ideal-world evaluation of f by a trusted party. More concretely, it is required that for every adversary $\mathcal{A}$, which attacks the real execution of the protocol, there exists an adversary Sim, also referred to as a simulator, which can *achieve the same effect* in the ideal-world. Since we are in the programmable random oracle model, in the ideal world we allow the simulator to answer oracle queries however it chooses. Let's denote $\mathbf{x} = (x_1, \ldots, x_n)$.

The real execution In the real execution of the n-party protocol π for computing f is executed in the presence of an adversary $\mathcal{A}$. The honest parties follow the instructions of π. The adversary $\mathcal{A}$ takes as input the security parameter k, the set $I \subset [n]$ of corrupted parties and the inputs of the corrupted parties. $\mathcal{A}$ sends all messages in place of corrupted parties and may follow an arbitrary polynomial-time strategy. The adversary and all other parties can interact with the random oracle RO.

The interaction of $\mathcal{A}$ with a protocol π defines a random variable $\mathsf{REAL}_{\pi,\mathcal{A}^{\mathsf{RO}},I}(k, \mathbf{x})$ whose value is determined by the coin tosses of the adversary and the honest parties. This random variable contains the output of the adversary (which may be an arbitrary function of its view) as well as the outputs of the honest parties. We let $\mathsf{REAL}_{\pi,\mathcal{A}^{\mathsf{RO}},I}$ denote the distribution ensemble $\{\mathsf{REAL}_{\pi,\mathcal{A}^{\mathsf{RO}},I}(k, \mathbf{x})\}_{k \in \mathsf{N}, \langle \mathbf{x}, z \rangle \in \{0,1\}^*}$.

The ideal execution . An ideal execution for a function f proceeds as follows, with an adversary Sim which attempts to mimic the behavior of $\mathcal{A}$. Note that in the ideal execution, there is no explicit random oracle and instead, Sim *can internally emulate* all queries made to the random oracle.

- **Send inputs to the trusted party:** The parties send their inputs to the trusted party, and we let x_i' denote the value sent by P_i.
- **Trusted party sends output to the adversary:** The trusted party computes $f(x_1', \ldots, x_n') = (y_1, \ldots, y_n)$ and sends $\{y_i\}_{i \in I}$ to the adversary Sim.
- **Adversary instructs trusted party to abort or continue:** This is formalized by having the adversary send either a continue or abort message to the trusted party. In the latter case, the trusted party sends to each uncorrupted party P_i its output value y_i. In the former case, the trusted party sends the special symbol $\perp$ to each uncorrupted party.
- **Outputs:** Sim outputs an arbitrary function of its view, and the honest parties output the values obtained from the trusted party.

The interaction of Sim with the trusted party defines a random variable $\mathsf{IDEAL}_{f_\perp,\mathsf{Sim}}(k, \mathbf{x})$ as above, and we let $\mathsf{IDEAL}_{f_\perp,\mathsf{Sim}} = \{\mathsf{IDEAL}_{f_\perp,\mathsf{Sim},I}(k, \mathbf{x})\}_{k \in \mathsf{N}, \langle \mathbf{x} \rangle \in \{0,1\}^*}$ where the subscript "$\perp$" indicates that the adversary can abort computation of f. We emphasize that the exclusion of the random oracle in the ideal world is intentional; adding it would give Sim no extra power.

Having defined the real and the ideal worlds, we now proceed to define our notion of security.

Definition A.1. Let k be the security parameter. Let f be an n-party randomized functionality, and π be an n-party protocol for $n \in \mathsf{N}$. We say π t-securely computes f *with abort* in the presence of malicious adversaries if for every PPT adversary $\mathcal{A}$ there exists a polynomial time adversary Sim such that for any $I \subset [n]$ with $|I| \leq t$ and any PPT distinguisher $\mathcal{D}$ the following quantity is negligible:

$$|Pr[\mathcal{D}(\mathsf{REAL}_{\pi,\mathsf{RO},\mathcal{A}^{\mathsf{RO}},I}(k, \mathbf{x})) = 1] - Pr[\mathcal{D}(\mathsf{IDEAL}_{f_\perp,\mathsf{Sim}}(k, \mathbf{x})) = 1]|.$$

Note that since we are in the programmable random oracle model where the distinguisher $\mathcal{D}$ doesn't get access to the random oracle.

B Further Preliminaries

B.1 Puncturable Pseudorandom Functions

We use the definition of puncturable PRFs given in [SW14], given as follows.

A puncturable family of PRFs F is given by a triple of turing machines PPRF.KeyGen, PPRF.Puncture, and F, and a pair of computable functions $n()$ and $m()$, satisfying the following conditions:

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

$$\Pr\left[F(K,x) = F(K_S,x) \mid \begin{array}{l} K \leftarrow \mathsf{PPRF.KeyGen}(1^\lambda), \\ K_S \leftarrow \mathsf{PPRF.Puncture}(K,S) \end{array}\right] = 1$$

- **Pseudorandom at punctured points:** For every PPT adversary (A_1, A_2) such that $A_1(1^\lambda)$ outputs a set $S \subseteq \{0,1\}^{n(\lambda)}$ and state σ, consider an experiment where $K \leftarrow \mathsf{PPRF.KeyGen}(1^\lambda)$ and $K_S \leftarrow \mathsf{PPRF.Puncture}(K,S)$. Then we have

$$\left|\Pr\left[A_2(\sigma, K_S, S, F(K,S)) = 1\right] - \Pr\left[A_2(\sigma, K_S, S, U_{m(\lambda)|S|}) = 1\right]\right| = negl(\lambda)$$

  where $F(K,S)$ denotes the concatenation of $F(K,x)$ for all $x \in S$ in lexicographic order and U_ℓ denotes the uniform distribution over ℓ bits.

B.2 Somewhere Statistically Binding Hash

Hubáček and Wichs [HW15] introduced the notion of Somewhere Statistically Binding (SSB) hash. It is parameterized by polynomials $\ell_{\text{fn}}, \ell_{\text{hash}}, \ell_{\text{open}}$, consists of algorithms $\mathsf{Gen\text{-}Enf}, \mathsf{Open}, \mathsf{Verify}$ with the following syntax.

$\mathsf{Gen\text{-}Enf}(1^\lambda, L, i)$: The setup algorithm takes as input security parameter λ (in unary), input length L (in binary) and index $i \in \{1,\dots,L\}$ (in binary). It outputs a polynomial time hash function $H : \{0,1\}^L \to \{0,1\}^{\ell_{\text{hash}}}$ whose description is at most $\ell_{\text{fn}}(\lambda)$ bits long.

$\mathsf{Open}(H, x, j)$: The opening algorithm takes as input the description of a hash function H, string $x \in \{0,1\}^L$ and an index $j \in \{1,\dots,L\}$. It outputs an opening $\pi \in \{0,1\}^{\ell_{\text{open}}}$.

$\mathsf{Verify}(H, h, j, u, \pi)$: The verification algorithm takes as input a hash function H, a hash value $h \in \{0,1\}^{\ell_{\text{hash}}}$, index $j \in \{1,\dots,L\}$, bit $u \in \{0,1\}$ and a proof $\pi \in \{0,1\}^{\ell_{\text{open}}}$. It outputs 0/1.

Definition B.1. An SSB hash scheme $\mathsf{SSB} = (\mathsf{Gen}, \mathsf{Open}, \mathsf{Verify})$, parameterized by $\ell_{\text{fn}}, \ell_{\text{hash}}, \ell_{\text{open}}$, is said to be secure if it satisfies the following correctness, statistically binding and index hiding properties:

- Correctness: For all $\lambda \in \mathbb{N}$, $L \leq 2^\lambda$, indices $i, j \in \{1,\dots,L\}$, string $x \in \{0,1\}^L$, if $H \leftarrow \mathsf{Gen\text{-}Enf}(1^\lambda, L, i)$ and $\pi \leftarrow \mathsf{Open}(H, x, j)$, then $\mathsf{Verify}(H, H(x), j, x[j], \pi) = 1$.
- Somewhere Statistically Binding Property: For any security parameter $\lambda \in \mathbb{N}$, $L \leq 2^\lambda$, index $i \in \{1,\dots,L\}$, there does not exist hash value h, distinct values $u \neq u'$ and openings π, π' such that $\mathsf{Verify}(H, h, i, u, \pi) = \mathsf{Verify}(H, h, i, u', \pi')$.
- Index Hiding Property: For every PPT algorithm $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a negligible function negl$(\cdot)$ such that for all $\lambda \in \mathbb{N}$, $\Pr[1 \leftarrow \mathsf{Expt\text{-}SSB}_{\mathsf{SSB},\mathcal{A}}(\lambda)] \leq 1/2 + \text{negl}(\lambda)$, where $\mathsf{Expt\text{-}SSB}$ is defined in Figure 14.

Experiment $\mathsf{Expt\text{-}SSB}_{\mathsf{SSB},\mathcal{A}}(\lambda)$

- $(L, i_0, i_1, \sigma) \leftarrow \mathcal{A}_1(1^\lambda)$. If $L > 2^\lambda$, experiment outputs a uniformly random bit.
- $b \leftarrow \{0, 1\}$, $H \leftarrow \mathsf{Gen\text{-}Enf}(1^\lambda, L, i_b)$.
- $b' \leftarrow \mathcal{A}_2(H, \sigma)$. If $b = b'$, experiment outputs 1.

Figure 14: Index Hiding Experiment for SSB Hash

B.3 Indistinguishability Obfuscation for Circuits

Let $\mathcal{C}$ be a class of boolean circuits. An obfuscation scheme for $\mathcal{C}$ consists of two algorithms $\mathsf{Ckt.Obf}, \mathsf{Ckt.Eval}$ with the following syntax.

$\mathsf{Ckt.Obf}(C \in \mathcal{C}, 1^\lambda)$: The obfuscation algorithm is a PPT algorithm that takes as input a circuit $C \in \mathcal{C}$, security parameter λ (in unary). It outputs an obfuscated circuit $\widetilde{C}$.

$\mathsf{Ckt.Eval}(\widetilde{C}, x)$: The evaluation algorithm is a PPT algorithm takes as input an obfuscated circuit $\widetilde{C}$, an input $x \in \{0, 1\}^*$, and outputs $y \in \{0, 1, \perp\}$.

Definition B.2. An obfuscation scheme $\mathcal{O} = (\mathsf{Ckt.Obf}, \mathsf{Ckt.Eval})$ is said to be a secure indistingushability obfuscator for $\mathcal{C}$ if it satisfies the following correctness and security properties:

- Correctness: For every security parameter $\lambda \in \mathbb{N}$, input length $n \in \mathbb{N}$, circuit $C \in \mathcal{C}$ that takes n bit inputs, input $x \in \{0, 1\}^n$, $\mathsf{Ckt.Eval}(\mathsf{Ckt.Obf}(C, 1^\lambda), x) = C(x)$.
- Security: For every PPT adversary $\mathcal{A} = (A_1, A_2)$, there exists a negligible function $\text{negl}(\cdot)$ such that for all security parameters $\lambda \in \mathbb{N}$, $\Pr[1 \leftarrow \mathsf{Expt\text{-}iO\text{-}ckt}_{\mathcal{O},\mathcal{A}}(\lambda)] - 1/2 \leq \text{negl}(\lambda)$, where $\mathsf{Expt\text{-}iO\text{-}ckt}$ is defined in Figure 15.

Experiment $\mathsf{Expt\text{-}iO\text{-}ckt}_{\mathcal{O},\mathcal{A}}(\lambda)$

– $(C_0, C_1, \sigma) \leftarrow \mathcal{A}_1(1^\lambda)$

– If $|C_0| \neq |C_1|$, or if either C_0 or C_1 have different input lengths, then the experiment outputs a uniformly random bit.
Else, let n denote the input lengths of C_0, C_1. If there exists an input $x \in \{0, 1\}^n$ such that $C_0(x) \neq C_1(x)$, then the experiment outputs a uniformly random bit.

– $b \leftarrow \{0, 1\}$, $\widetilde{C} \leftarrow \mathsf{Ckt.Obf}(C_b, 1^\lambda)$.

– $b' \leftarrow \mathcal{A}_2(\sigma, \widetilde{C})$.

– Experiment outputs 1 if $b = b'$, else it outputs 0.

Figure 15: Security Experiment for Indistinguishability Obfuscation

B.4 Succinct Randomized Encodings

The notion of succinct randomized encodings was introduced by the works of [BGL+15, KLW15, CHJV15]. In a randomized encoding scheme, we have an encoding algorithm and a decoding algorithm. The encoding algorithm takes as input a Turing machine M, an input x and outputs an encoding of (M, x). The decoding algorithm can use this encoding to compute $M(x)$. Here, it is important that the time to encode only depends on the size of M, x and the security parameter. In particular, it must be independent of the running time of M on input x. In addition to the encoding algorithms being succinct, we will also require a *succinct simulator*. While the previous works did not require this property, the construction of [KLW15] satisfies this property.

Let $\mathcal{M}$ be a family of oblivious boolean Turing machines. A randomized encoding scheme for $\mathcal{M}$ consists of algorithms Encode and Decode with the following syntax.

$\mathsf{Encode}((M, \mathsf{tmf}(\cdot)), x, T, 1^\lambda)$: The encoding algorithm is a PPT algorithm that takes as input an oblivious Turing machine M with tape movement function $\mathsf{tmf}(\cdot)$, input x, security parameter λ (in unary) and time bound $T \leq 2^\lambda$ (in binary). It outputs an encoding $\widetilde{M_x}$ of size $\mathsf{poly}(|M|, |x|, \log T, \lambda)$.

$\mathsf{Decode}(\widetilde{M_x})$: The decoding algorithm takes as input an encoding $\widetilde{M_x}$ and outputs $y \in \{0, 1, \perp\}$.

Definition B.3. Let $\mathcal{M}$ be a family of Turing machines. A randomized encoding scheme $\mathsf{SRE} = (\mathsf{Encode}, \mathsf{Decode})$ is said to be a succinct randomized encoding scheme if it satisfies the following correctness, efficiency and security properties.

- Correctness: For every machine $M \in \mathcal{M}$ with tape movement function $\mathsf{tmf}(\cdot)$, input $x \in \{0,1\}^*$, security parameter λ and T,

$$\mathsf{Decode}(\mathsf{Encode}((M, \mathsf{tmf}(\cdot)), x, T, 1^\lambda)) = \mathsf{TM}(M, x, T).$$

- Efficiency: There exist polynomials p_1 and p_2 such that for every machine $M \in \mathcal{M}$, $x \in \{0,1\}^*$, $T \in \mathbb{N}$ and $\lambda \in \mathbb{N}$, the time to encode $\widetilde{M_x} \leftarrow \mathsf{Encode}((M, \mathsf{tmf}(\cdot)), x, T, 1^\lambda)$ is bounded by $p_1(M, x, \log T, \lambda)$, and the time to decode $\widetilde{M_x}$ is bounded by $\min(\mathsf{Time}(M, x), T) \cdot p_2(\lambda, \log T)$.

- Security: For every PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a PPT simulator $\mathcal{S}$ such that for all PPT distinguishers $\mathcal{D}$, there exists a negligible function negl$(\cdot)$ such that for all $\lambda \in \mathbb{N}$, $\Pr[1 \leftarrow \mathcal{D}(\mathsf{Expt\text{-}SRE\text{-}Real}_{\mathsf{SRE},\mathcal{A}}(\lambda))] - \Pr[1 \leftarrow \mathcal{D}(\mathsf{Expt\text{-}SRE\text{-}Ideal}_{\mathsf{SRE},\mathcal{A},\mathcal{S}}(\lambda))] \leq \text{negl}(\lambda)$, where Expt-SRE-Real and Expt-SRE-Ideal are defined in Figure 16. Moreover, the running time of $\mathcal{S}$ is bounded by some polynomial $p_{\mathcal{S}}(|M|, |x_1|, \log T, \lambda)$.

Experiments $\mathsf{Expt\text{-}SRE\text{-}Real}_{\mathsf{OcRE},\mathcal{A}}(\lambda)$ and $\mathsf{Expt\text{-}SRE\text{-}Ideal}_{\mathsf{OcRE},\mathcal{A},\mathcal{S}}(\lambda)$

$\mathsf{Expt\text{-}SRE\text{-}Real}_{\mathsf{SRE},\mathcal{A}}(\lambda)$:

- $((M, \mathsf{tmf}(\cdot)), x, T, \sigma) \leftarrow \mathcal{A}_1(1^\lambda)$.[a]
- $\widetilde{M} \leftarrow \mathsf{Encode}((M, \mathsf{tmf}(\cdot))x, T, 1^\lambda)$.
- Experiment outputs $\mathcal{A}_2(\widetilde{M}, \sigma)$.

[a]The adversary $\mathcal{A}_1$ outputs an oblivious Turing machine M with tape movement function $\mathsf{tmf}(\cdot)$.

$\mathsf{Expt\text{-}SRE\text{-}Ideal}_{\mathsf{SRE},\mathcal{A},\mathcal{S}}(\lambda)$:

- $((M, \mathsf{tmf}(\cdot)), x, T, \sigma) \leftarrow \mathcal{A}_1(1^\lambda)$. [a]
- Let $t^* = \min(T, \mathsf{Time}(M, x))$ and $b^* = \mathsf{TM}(M, x, T)$.
- $\widetilde{M} \leftarrow \mathcal{S}(1^{|M|}, 1^{|x|}, t^*, b^*, \mathsf{tmf}(\cdot), 1^\lambda)$.[b]
- Experiment outputs $\mathcal{A}_2(\widetilde{M}, \sigma)$.

[a]The adversary $\mathcal{A}_1$ outputs an oblivious Turing machine M with tape movement function $\mathsf{tmf}(\cdot)$.

[b]Note that since $\mathcal{S}$ is a polynomial time simulator, the running time of $\mathcal{S}$ is polynomial in $|M|$, $|x|$, λ, $\log T$.

Figure 16: Simulation Security Experiments for randomized encodings

C Primitives from [KLW15]

C.1 Notations

For our partial randomized encoding scheme, we will use the following notations/conventions for Turing machines. We will assume the Turing machine has a worktape and an 'auxiliary-input' tape. Both these tapes have a header, which is initially positioned at index 0 and moves left/right (denoted by $-1/+1$ respectively). The transition function δ takes as input the current state, symbol located at the head positions of the tape and aux-tape respectively, and outputs the new state, the new symbol to be written on tape (at the head position), and the movement for the tape header and aux-tape header respectively. More formally, it is defined as follows.

Turing machines A Turing machine is a 7-tuple $M = \langle Q, \Sigma_{\mathsf{tape}}, \Sigma_{\text{inp}}, \delta, q_0, q_{\text{acc}}, q_{\text{rej}} \rangle$ with the following semantics:

- Q is the set of states with start state q_0, accept state q_{acc} and reject state q_{rej}.
- Σ_{inp} is the set of inputs symbols
- Σ_{tape} is the set of tape symbols. We will assume $\Sigma_{\text{inp}} \subset \Sigma_{\mathsf{tape}}$ and there is a special blank symbol '␣' $\in \Sigma_{\mathsf{tape}} \setminus \Sigma_{\text{inp}}$.
- $\delta : Q \times \Sigma_{\mathsf{tape}} \times \Sigma_{\text{inp}} \to Q \times \Sigma_{\mathsf{tape}} \times \{+1, -1\} \times \{+1, -1\}$ is the transition function.

We will assume the computation begins with the work-tape containing the actual input x, the aux-tape contains the auxiliary input x_2, both heads positioned at index 0, and the starting state being q_0. The computation ends when the state reaches either q_{acc} or q_{rej}. If it reaches q_{acc}, then the machine M accepts (x, x_{aux}), and if it reaches q_{rej}, it rejects (x, x_{aux}).

C.2 iO-Compatible Primitives

In this section, we will define extensions of some cryptographic primitives that makes them 'compatible' with indistinguishability obfuscation. Here, we define the syntax and security notions for these primitives. Their constructions can be found in [KLW15].

C.2.1 Splittable Signatures

A splittable signature scheme is a normal signature scheme, augmented by some additional algorithms that produce alternative signing and verification keys with differing capabilities. More precisely, there will be 'all but one' keys that work correctly except for a single message m^*, and there will be 'one' keys that work only for m^*. There will also be a reject-verification key that always outputs reject when used to verify a signature.

Syntax A splittable signature scheme $\mathcal{S}$ for message space $\mathcal{M}$ consists of the following algorithms:

$\mathsf{Spl.Setup}(1^\lambda)$ The setup algorithm is a randomized algorithm that takes as input the security parameter λ and outputs a signing key SK, verification key VK and *reject-verification key* VK_{rej}.

$\mathsf{Spl.Sign}(\text{SK}, m)$ The signing algorithm is a deterministic algorithm that takes as input a signing key SK and a message $m \in \mathcal{M}$. It outputs a signature σ.

$\mathsf{Spl.Verify}(\text{VK}, m, \sigma)$ The verification algorithm is a deterministic algorithm that takes as input a verification key VK, signature σ and a message m. It outputs either 0 or 1.

$\mathsf{Spl.Split}(\text{SK}, m^*)$ The splitting algorithm is randomized. It takes as input a secret key SK and a message $m^* \in \mathcal{M}$. It outputs a signature $\sigma_{\mathsf{one}} = \mathsf{Spl.Sign}(\text{SK}, m^*)$, a one-message verification key VK_{one}, an all-but-one signing key SK_{abo} and an all-but-one verification key VK_{abo}.

$\mathsf{Spl.Sign\text{-}abo}(\text{SK}_{\text{abo}}, m)$ The all-but-one signing algorithm is deterministic. It takes as input an all-but-one signing key SK_{abo} and a message m, and outputs a signature σ.

Correctness Let $m^* \in \mathcal{M}$ be any message. Let $(\text{SK}, \text{VK}, \text{VK}_{\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$ and $(\sigma_{\mathsf{one}}, \text{VK}_{\mathsf{one}}, \text{SK}_{\text{abo}}, \text{VK}_{\text{abo}}) \leftarrow \mathsf{Spl.Split}(\text{SK}, m^*)$. Then, we require the following correctness properties:

1. For all $m \in \mathcal{M}$, $\mathsf{Spl.Verify}(\text{VK}, m, \mathsf{Spl.Sign}(\text{SK}, m)) = 1$.
2. For all $m \in \mathcal{M}, m \neq m^*$, $\mathsf{Spl.Sign}(\text{SK}, m) = \mathsf{Spl.Sign\text{-}abo}(\text{SK}_{\text{abo}}, m)$.

3. For all σ, $\mathsf{Spl.Verify}(\mathrm{VK}_{\mathsf{one}}, m^*, \sigma) = \mathsf{Spl.Verify}(\mathrm{VK}, m^*, \sigma)$.
4. For all $m \neq m^*$ and σ, $\mathsf{Spl.Verify}(\mathrm{VK}, m, \sigma) = \mathsf{Spl.Verify}(\mathrm{VK}_{\mathrm{abo}}, m, \sigma)$.
5. For all $m \neq m^*$ and σ, $\mathsf{Spl.Verify}(\mathrm{VK}_{\mathsf{one}}, m, \sigma) = 0$.
6. For all σ, $\mathsf{Spl.Verify}(\mathrm{VK}_{\mathrm{abo}}, m^*, \sigma) = 0$.
7. For all σ and all $m \in \mathcal{M}$, $\mathsf{Spl.Verify}(\mathrm{VK}_{\mathsf{rej}}, m, \sigma) = 0$.

Security We will now define the security notions for splittable signature schemes. Each security notion is defined in terms of a security game between a challenger and an adversary $\mathcal{A}$.

Definition C.1 ($\mathrm{VK}_{\mathsf{rej}}$ indistinguishability)**.** A splittable signature scheme $\mathcal{S}$ is said to be $\mathrm{VK}_{\mathsf{rej}}$ indistinguishable if any PPT adversary $\mathcal{A}$ has negligible advantage in the following security game:

$\mathsf{Exp}\text{-}\mathrm{VK}_{\mathsf{rej}}(1^\lambda, \mathcal{S}, \mathcal{A})$:

1. Challenger computes $(\mathrm{SK}, \mathrm{VK}, \mathrm{VK}_{\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$ .Next, it chooses $b \leftarrow \{0, 1\}$. If $b = 0$, it sends VK to $\mathcal{A}$. Else, it sends $\mathrm{VK}_{\mathsf{rej}}$.
2. $\mathcal{A}$ sends its guess b'.

$\mathcal{A}$ wins if $b = b'$.

We note that in the game above, $\mathcal{A}$ never receives any signatures and has no ability to produce them. This is why the difference between VK and $\mathrm{VK}_{\mathsf{rej}}$ cannot be tested.

Definition C.2 ($\mathrm{VK}_{\mathsf{one}}$ indistinguishability)**.** A splittable signature scheme $\mathcal{S}$ is said to be $\mathrm{VK}_{\mathsf{one}}$ indistinguishable if any PPT adversary $\mathcal{A}$ has negligible advantage in the following security game:

$\mathsf{Exp}\text{-}\mathrm{VK}_{\mathsf{one}}(1^\lambda, \mathcal{S}, \mathcal{A})$:

1. $\mathcal{A}$ sends a message $m^* \in \mathcal{M}$.
2. Challenger computes $(\mathrm{SK}, \mathrm{VK}, \mathrm{VK}_{\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$. Next, it computes ($\sigma_{\mathsf{one}}$, $\mathrm{VK}_{\mathsf{one}}$, $\mathrm{SK}_{\mathrm{abo}}$, $\mathrm{VK}_{\mathrm{abo}}$) $\leftarrow \mathsf{Spl.Split}(\mathrm{SK}, m^*)$. It chooses $b \leftarrow \{0, 1\}$. If $b = 0$, it sends $(\sigma_{\mathsf{one}}, \mathrm{VK}_{\mathsf{one}})$ to $\mathcal{A}$. Else, it sends $(\sigma_{\mathsf{one}}, \mathrm{VK})$ to $\mathcal{A}$.
3. $\mathcal{A}$ sends its guess b'.

$\mathcal{A}$ wins if $b = b'$.

We note that in the game above, $\mathcal{A}$ only receives the signature σ_{one} on m^*, on which VK and VK_{one} behave identically.

Definition C.3 ($\mathrm{VK}_{\mathrm{abo}}$ indistinguishability)**.** A splittable signature scheme $\mathcal{S}$ is said to be $\mathrm{VK}_{\mathrm{abo}}$ indistinguishable if any PPT adversary $\mathcal{A}$ has negligible advantage in the following security game:

$\mathsf{Exp}\text{-}\mathrm{VK}_{\mathrm{abo}}(1^\lambda, \mathcal{S}, \mathcal{A})$:

1. $\mathcal{A}$ sends a message $m^* \in \mathcal{M}$.
2. Challenger computes $(\mathrm{SK}, \mathrm{VK}, \mathrm{VK}_{\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$. Next, it computes ($\sigma_{\mathsf{one}}$, $\mathrm{VK}_{\mathsf{one}}$, $\mathrm{SK}_{\mathrm{abo}}$, $\mathrm{VK}_{\mathrm{abo}}$) $\leftarrow \mathsf{Spl.Split}(\mathrm{SK}, m^*)$. It chooses $b \leftarrow \{0, 1\}$. If $b = 0$, it sends $(\mathrm{SK}_{\mathrm{abo}}, \mathrm{VK}_{\mathrm{abo}})$ to $\mathcal{A}$. Else, it sends $(\mathrm{SK}_{\mathrm{abo}}, \mathrm{VK})$ to $\mathcal{A}$.
3. $\mathcal{A}$ sends its guess b'.

$\mathcal{A}$ wins if $b = b'$.

We note that in the game above, $\mathcal{A}$ does not receive or have the ability to create a signature on m^*. For all signatures $\mathcal{A}$ can create by signing with $\mathrm{SK}_{\mathrm{abo}}$, $\mathrm{VK}_{\mathrm{abo}}$ and VK will behave identically.

Definition C.4 (Splitting indistinguishability)**.** A splittable signature scheme $\mathcal{S}$ is said to be splitting indistinguishable if any PPT adversary $\mathcal{A}$ has negligible advantage in the following security game:

$\mathsf{Exp\text{-}Spl}(1^\lambda, \mathcal{S}, \mathcal{A})$:

1. $\mathcal{A}$ sends a message $m^* \in \mathcal{M}$.
2. Challenger computes $(\mathrm{SK}, \mathrm{VK}, \mathrm{VK}_{\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$, $(\mathrm{SK}', \mathrm{VK}', \mathrm{VK}'_{\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$. Next, it computes $(\sigma_{\mathsf{one}}, \mathrm{VK}_{\mathsf{one}}, \mathrm{SK}_{\mathrm{abo}}, \mathrm{VK}_{\mathrm{abo}}) \leftarrow \mathsf{Spl.Split}(\mathrm{SK}, m^*)$, $(\sigma'_{\mathsf{one}}, \mathrm{VK}'_{\mathsf{one}}, \mathrm{SK}'_{\mathrm{abo}}, \mathrm{VK}'_{\mathrm{abo}}) \leftarrow \mathsf{Spl.Split}(\mathrm{SK}', m^*)$. . It chooses $b \leftarrow \{0, 1\}$. If $b = 0$, it sends $(\sigma_{\mathsf{one}}, \mathrm{VK}_{\mathsf{one}}, \mathrm{SK}_{\mathrm{abo}}, \mathrm{VK}_{\mathrm{abo}})$ to $\mathcal{A}$. Else, it sends $(\sigma'_{\mathsf{one}}, \mathrm{VK}'_{\mathsf{one}}, \mathrm{SK}_{\mathrm{abo}}, \mathrm{VK}_{\mathrm{abo}})$ to $\mathcal{A}$.
3. $\mathcal{A}$ sends its guess b'.

$\mathcal{A}$ wins if $b = b'$.

In the game above, $\mathcal{A}$ is either given a system of $\sigma_{\mathsf{one}}, \mathrm{VK}_{\mathsf{one}}, \mathrm{SK}_{\mathrm{abo}}, \mathrm{VK}_{\mathrm{abo}}$ generated together by one call of $\mathsf{Spl.Setup}$ or a "split" system of $(\sigma'_{\mathsf{one}}, \mathrm{VK}'_{\mathsf{one}}, \mathrm{SK}_{\mathrm{abo}}, \mathrm{VK}_{\mathrm{abo}})$ where the all but one keys are generated separately from the signature and key for the one message m^*. Since the correctness conditions do not link the behaviors for the all but one keys and the one message values, this split generation is not detectable by testing verification for the σ_{one} that $\mathcal{A}$ receives or for any signatures that $\mathcal{A}$ creates honestly by signing with $\mathrm{SK}_{\mathrm{abo}}$.

C.2.2 Positional Accumulators

A positional accumulator for message space $\mathcal{M}_\lambda$ consists of the following algorithms.

- $\mathsf{Acc.Setup}(1^\lambda, T) \rightarrow (\mathsf{P}, \mathsf{acc}_0, \textsc{store}_0)$ The setup algorithm takes as input a security parameter λ in unary and an integer T in binary representing the maximum number of values that can stored. It outputs public parameters P, an initial accumulator value acc_0, and an initial storage value $\textsc{store}_0$.

- $\mathsf{Acc.Setup\text{-}Enforce\text{-}Read}(1^\lambda, T, (m_1, \textsc{index}_1), \ldots, (m_k, \textsc{index}_k), \textsc{index}^*) \rightarrow (\mathsf{P}, \mathsf{acc}_0, \textsc{store}_0)$: The setup enforce read algorithm takes as input a security parameter λ in unary, an integer T in binary representing the maximum number of values that can be stored, and a sequence of symbol, index pairs, where each index is between 0 and $T-1$, and an additional $\textsc{index}^*$ also between 0 and $T-1$. It outputs public parameters P, an initial accumulator value acc_0, and an initial storage value $\textsc{store}_0$.

- $\mathsf{Acc.Setup\text{-}Enforce\text{-}Write}(1^\lambda, T, (m_1, \textsc{index}_1), \ldots, (m_k, \textsc{index}_k)) \rightarrow (\mathsf{P}, \mathsf{acc}_0, \textsc{store}_0)$: The setup enforce write algorithm takes as input a security parameter λ in unary, an integer T in binary representing the maximum number of values that can be stored, and a sequence of symbol, index pairs, where each index is between 0 and $T-1$. It outputs public parameters P, an initial accumulator value acc_0, and an initial storage value $\textsc{store}_0$.

- $\mathsf{Acc.Prep\text{-}Read}(\mathsf{P}, \textsc{store}_{in}, \textsc{index}) \rightarrow (m, \pi)$: The prep-read algorithm takes as input the public parameters P, a storage value $\textsc{store}_{in}$, and an index between 0 and $T-1$. It outputs a symbol m (that can be ϵ) and a value π.

- $\mathsf{Acc.Prep\text{-}Write}(\mathsf{P}, \textsc{store}_{in}, \textsc{index}) \rightarrow aux$: The prep-write algorithm takes as input the public parameters P, a storage value $\textsc{store}_{in}$, and an index between 0 and $T-1$. It outputs an auxiliary value aux.

- $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}, \mathsf{acc}_{in}, m_{read}, \textsc{index}, \pi) \rightarrow \{True, False\}$: The verify-read algorithm takes as input the public parameters P, an accumulator value acc_{in}, a symbol, m_{read}, an index between 0 and $T-1$, and a value π. It outputs $True$ or $False$.

- $\mathsf{Acc.Write\text{-}Store}(\mathsf{P}, \textsc{store}_{in}, \textsc{index}, m) \rightarrow \textsc{store}_{out}$: The write-store algorithm takes in the public parameters, a storage value $\textsc{store}_{in}$, an index between 0 and $T-1$, and a symbol m. It outputs a storage value $\textsc{store}_{out}$.
- $\mathsf{Update}(\mathsf{P}, \mathsf{acc}_{in}, m_{write}, \textsc{index}, aux) \rightarrow \mathsf{acc}_{out}$ or $Reject$: The update algorithm takes in the public parameters P, an accumulator value acc_{in}, a symbol m_{write}, and index between 0 and $T-1$, and an auxiliary value aux. It outputs an accumulator value acc_{out} or $Reject$.

In general we will think of the $\mathsf{Acc.Setup}$ algorithm as being randomized and the other algorithms as being deterministic. However, one could consider non-deterministic variants.

Correctness We consider any sequence $(m_1, \textsc{index}_1), \ldots, (m_k, \textsc{index}_k)$ of symbols $m_1, \ldots, m_k$ and indices $\textsc{index}_1, \ldots, \textsc{index}_k$ each between 0 and $T-1$. We fix any $\mathsf{P}, \mathsf{acc}_0, \textsc{store}_0 \leftarrow \mathsf{Acc.Setup}(1^\lambda, T)$. For j from 1 to k, we define $\textsc{store}_j$ iteratively as $\textsc{store}_j := \mathsf{Acc.Write\text{-}Store}(\mathsf{P}, \textsc{store}_{j-1}, \textsc{index}_j, m_j)$. We similarly define aux_j and acc_j iteratively as $aux_j := \mathsf{Acc.Prep\text{-}Write}(\mathsf{P}, \textsc{store}_{j-1}, \textsc{index}_j)$ and $\mathsf{acc}_j := Update(\mathsf{P}, \mathsf{acc}_{j-1}, m_j, \textsc{index}_j, aux_j)$. Note that the algorithms other than $\mathsf{Acc.Setup}$ are deterministic, so these definitions fix precise values, not random values (conditioned on the fixed starting values $\mathsf{P}, \mathsf{acc}_0, \textsc{store}_0$).

We require the following correctness properties:

1. For every $\textsc{index}$ between 0 and $T-1$, $\mathsf{Acc.Prep\text{-}Read}(\mathsf{P}, \textsc{store}_k, \textsc{index})$ returns m_i, π, where i is the largest value in $[k]$ such that $\textsc{index}_i = \textsc{index}$. If no such value exists, then $m_i = \epsilon$.
2. For any $\textsc{index}$, let $(m, \pi) \leftarrow \mathsf{Acc.Prep\text{-}Read}(\mathsf{P}, \textsc{store}_k, \textsc{index})$. Then $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}, \mathsf{acc}_k, m, \textsc{index}, \pi) = True$.

Remarks on Efficiency In our construction, all algorithms will run in time polynomial in their input sizes. More precisely, $\mathsf{Acc.Setup}$ will be polynomial in λ and $\log(T)$. Also, accumulator and π values should have size polynomial in λ and $\log(T)$, so $\mathsf{Acc.Verify\text{-}Read}$ and Update will also run in time polynomial in λ and $\log(T)$. Storage values will have size polynomial in the number of values stored so far. $\mathsf{Acc.Write\text{-}Store}$, $\mathsf{Acc.Prep\text{-}Read}$, and $\mathsf{Acc.Prep\text{-}Write}$ will run in time polynomial in λ and T.

Security Let Acc = ($\mathsf{Acc.Setup}$, $\mathsf{Acc.Setup\text{-}Enforce\text{-}Read}$, $\mathsf{Acc.Setup\text{-}Enforce\text{-}Write}$, $\mathsf{Acc.Prep\text{-}Read}$, $\mathsf{Acc.Prep\text{-}Write}$, $\mathsf{Acc.Verify\text{-}Read}$, $\mathsf{Acc.Write\text{-}Store}$, Update) be a positional accumulator for symbol set $\mathcal{M}$. We require Acc to satisfy the following notions of security.

Definition C.5 (Indistinguishability of Read Setup)**.** A positional accumulator Acc is said to satisfy indistinguishability of read setup if any PPT adversary $\mathcal{A}$'s advantage in the security game $\mathsf{Exp\text{-}Setup\text{-}Acc}(1^\lambda, \mathsf{Acc}, \mathcal{A})$ is at most negligible in λ, where $\mathsf{Exp\text{-}Setup\text{-}Acc}$ is defined as follows.

$\mathsf{Exp\text{-}Setup\text{-}Acc}(1^\lambda, \mathsf{Acc}, \mathcal{A})$

1. Adversary chooses a bound $T \in \Theta(2^\lambda)$ and sends it to challenger.
2. $\mathcal{A}$ sends k messages $m_1, \ldots, m_k \in \mathcal{M}$ and k indices $\textsc{index}_1, \ldots,$ $indexA_k \in \{0, \ldots, T-1\}$ to the challenger.
3. The challenger chooses a bit b. If $b = 0$, the challenger outputs $(\mathsf{P}, \mathsf{acc}_0, \textsc{store}_0) \leftarrow \mathsf{Acc.Setup}(1^\lambda, T)$. Else, it outputs $(\mathsf{P}, \mathsf{acc}_0, \textsc{store}_0) \leftarrow \mathsf{Acc.Setup\text{-}Enforce\text{-}Read}(1^\lambda, T, (m_1, \textsc{index}_1), \ldots, (m_k, \textsc{index}_k))$.
4. $\mathcal{A}$ sends a bit b'.

$\mathcal{A}$ wins the security game if $b = b'$.

Definition C.6 (Indistinguishability of Write Setup)**.** A positional accumulator Acc is said to satisfy indistinguishability of write setup if any PPT adversary $\mathcal{A}$'s advantage in the security game

$\mathsf{Exp\text{-}Setup\text{-}Acc}(1^\lambda, \mathsf{Acc}, \mathcal{A})$ is at most negligible in λ, where $\mathsf{Exp\text{-}Setup\text{-}Acc}$ is defined as follows.

$\mathsf{Exp\text{-}Setup\text{-}Acc}(1^\lambda, \mathsf{Acc}, \mathcal{A})$

1. Adversary chooses a bound $T \in \Theta(2^\lambda)$ and sends it to challenger.
2. $\mathcal{A}$ sends k messages $m_1, \ldots, m_k \in \mathcal{M}$ and k indices $\textsc{index}_1, \ldots,$ $indexA_k \in \{0, \ldots, T-1\}$ to the challenger.
3. The challenger chooses a bit b. If $b = 0$, the challenger outputs $(\mathsf{P}, \mathsf{acc}_0, \textsc{store}_0) \leftarrow \mathsf{Acc.Setup}(1^\lambda, T)$. Else, it outputs $(\mathsf{P}, \mathsf{acc}_0, \textsc{store}_0) \leftarrow \mathsf{Acc.Setup\text{-}Enforce\text{-}Write}(1^\lambda, T, (m_1, \textsc{index}_1), \ldots, (m_k, \textsc{index}_k))$.
4. $\mathcal{A}$ sends a bit b'.

$\mathcal{A}$ wins the security game if $b = b'$.

Definition C.7 (Read Enforcing)**.** Consider any $\lambda \in \mathbb{N}$, $T \in \Theta(2^\lambda)$, $m_1, \ldots, m_k \in \mathcal{M}$, $\textsc{index}_1, \ldots, \textsc{index}_k \in \{0, \ldots, T-1\}$ and any $\textsc{index}^* \in \{0, \ldots, T-1\}$.

Let $(\mathsf{P}, \mathsf{acc}_0, \textsc{store}_0) \leftarrow \mathsf{Acc.Setup\text{-}Enforce\text{-}Read}(1^\lambda, T, (m_1, \textsc{index}_1), \ldots, (m_k, \textsc{index}_k), \textsc{index}^*)$. For j from 1 to k, we define $\textsc{store}_j$ iteratively as $\textsc{store}_j := \mathsf{Acc.Write\text{-}Store}(\mathsf{P}, \textsc{store}_{j-1}, \textsc{index}_j, m_j)$. We similarly define aux_j and acc_j iteratively as $aux_j := \mathsf{Acc.Prep\text{-}Write}(\mathsf{P}, \textsc{store}_{j-1}, \textsc{index}_j)$ and $\mathsf{acc}_j := Update(\mathsf{P}, \mathsf{acc}_{j-1}, m_j, \textsc{index}_j, aux_j)$. Acc is said to be *read enforcing* if $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}, \mathsf{acc}_k, m, \textsc{index}^*, \pi) = True$, then either $\textsc{index}^* \notin \{\textsc{index}_1, \ldots, \textsc{index}_k\}$ and $m = \epsilon$, or $m = m_i$ for the largest $i \in [k]$ such that $\textsc{index}_i = \textsc{index}^*$. Note that this is an information-theoretic property: we are requiring that for all other symobls m, values of π that would cause $\mathsf{Acc.Verify\text{-}Read}$ to output $True$ at $\textsc{index}^*$ do no exist.

Definition C.8 (Write Enforcing)**.** Consider any $\lambda \in \mathbb{N}$, $T \in \Theta(2^\lambda)$, $m_1, \ldots, m_k \in \mathcal{M}$, $\textsc{index}_1, \ldots, \textsc{index}_k \in \{0, \ldots, T-1\}$. Let $(\mathsf{P}, \mathsf{acc}_0, \textsc{store}_0) \leftarrow \mathsf{Acc.Setup\text{-}Enforce\text{-}Write}(1^\lambda, T, (m_1, \textsc{index}_1), \ldots, (m_k, \textsc{index}_k))$. For j from 1 to k, we define $\textsc{store}_j$ iteratively as $\textsc{store}_j := \mathsf{Acc.Write\text{-}Store}(\mathsf{P}, \textsc{store}_{j-1}, \textsc{index}_j, m_j)$. We similarly define aux_j and acc_j iteratively as $aux_j := \mathsf{Acc.Prep\text{-}Write}(\mathsf{P}, \textsc{store}_{j-1}, \textsc{index}_j)$ and $\mathsf{acc}_j := Update(\mathsf{P}, \mathsf{acc}_{j-1}, m_j, \textsc{index}_j, aux_j)$. Acc is said to be *write enforcing* if $\mathsf{Update}(\mathsf{P}, \mathsf{acc}_{k-1}, m_k, \textsc{index}_k, aux) = \mathsf{acc}_{out} \neq Reject$, for any aux, then $\mathsf{acc}_{out} = \mathsf{acc}_k$. Note that this is an information-theoretic property: we are requiring that an aux value producing an accumulated value other than acc_k or $Reject$ does not exist.

C.2.3 Iterators

In this section, we define the notion of *cryptographic iterators*. A cryptographic iterator essentially consists of a small state that is updated in an iterative fashion as messages are received. An update to apply a new message given current state is performed via some public parameters.

Since states will remain relatively small regardless of the number of messages that have been iteratively applied, there will in general be many sequences of messages that can lead to the same state. However, our security requirement will capture that the normal public parameters are computationally indistinguishable from specially constructed "enforcing" parameters that ensure that a particular *single* state can be only be obtained as an output as an update to precisely one other state, message pair. Note that this enforcement is a very localized property to a particular state, and hence can be achieved information-theoretically when we fix ahead of time where exactly we want this enforcement to be.

Syntax Let ℓ be any polynomial. An iterator $\mathcal{I}$ with message space $\mathcal{M}_\lambda = \{0,1\}^{\ell(\lambda)}$ and state space $\mathcal{S}_\lambda$ consists of three algorithms - $\mathsf{Itr.Setup}$, $\mathsf{Itr.Setup\text{-}Enforce}$ and $\mathsf{Itr.Iterate}$ defined below.

$\mathsf{Itr.Setup}(1^\lambda, T)$ The setup algorithm takes as input the security parameter λ (in unary), and an integer bound T (in binary) on the number of iterations. It outputs public parameters P and an initial state $v_0 \in \mathcal{S}_\lambda$.

$\mathsf{Itr.Setup\text{-}Enforce}(1^\lambda, T, \mathbf{m} = (m_1, \ldots, m_k))$ The enforced setup algorithm takes as input the security parameter λ (in unary), an integer bound T (in binary) and k messages $(m_1, \ldots, m_k)$, where each $m_i \in \{0,1\}^{\ell(\lambda)}$ and k is some polynomial in λ. It outputs public parameters P and a state $v_0 \in \mathcal{S}$.

$\mathsf{Itr.Iterate}(\mathsf{P}, v_{\mathsf{in}}, m)$ The iterate algorithm takes as input the public parameters P, a state v_{in}, and a message $m \in \{0,1\}^{\ell(\lambda)}$. It outputs a state $v_{\mathsf{out}} \in \mathcal{S}_\lambda$.

For simplicity of notation, we will drop the dependence of ℓ on λ. Also, for any integer $k \le T$, we will use the notation $\mathsf{Itr.Iterate}^k(\mathsf{P}, v_0, (m_1, \dots, m_k))$ to denote $\mathsf{Itr.Iterate}(\mathsf{P}, v_{k-1}, m_k)$, where $v_j = \mathsf{Itr.Iterate}(\mathsf{P}, v_{j-1}, m_j)$ for all $1 \le j \le k-1$.

Security Let $\mathcal{I} = (\mathsf{Itr.Setup}, \mathsf{Itr.Setup\text{-}Enforce}, \mathsf{Itr.Iterate})$ be an interator with message space $\{0,1\}^\ell$ and state space $\mathcal{S}_\lambda$. We require the following notions of security.

Definition C.9 (Indistinguishability of Setup)**.** An iterator $\mathcal{I}$ is said to satisfy indistinguishability of Setup phase if any PPT adversary $\mathcal{A}$'s advantage in the security game $\mathsf{Exp\text{-}Setup\text{-}Itr}(1^\lambda, \mathcal{I}, \mathcal{A})$ at most is negligible in λ, where $\mathsf{Exp\text{-}Setup\text{-}Itr}$ is defined as follows.

$\mathsf{Exp\text{-}Setup\text{-}Itr}(1^\lambda, \mathcal{I}, \mathcal{A})$

1. The adversary $\mathcal{A}$ chooses a bound $T \in \Theta(2^\lambda)$ and sends it to challenger.
2. $\mathcal{A}$ sends k messages $m_1, \dots, m_k \in \{0,1\}^\ell$ to the challenger.
3. The challenger chooses a bit b. If $b = 0$, the challenger outputs $(\mathsf{P}, v_0) \leftarrow \mathsf{Itr.Setup}(1^\lambda, T)$. Else, it outputs $(\mathsf{P}, v_0) \leftarrow \mathsf{Itr.Setup\text{-}Enforce}(1^\lambda, T, 1^k, \mathbf{m} = (m_1, \dots, m_k))$.
4. $\mathcal{A}$ sends a bit b'.

$\mathcal{A}$ wins the security game if $b = b'$.

Definition C.10 (Enforcing)**.** Consider any $\lambda \in \mathbb{N}$, $T \in \Theta(2^\lambda)$, $k < T$ and $m_1, \dots, m_k \in \{0,1\}^\ell$. Let $(\mathsf{P}, v_0) \leftarrow \mathsf{Itr.Setup\text{-}Enforce}(1^\lambda, T, \mathbf{m} = (m_1, \dots, m_k))$ and $v_j = \mathsf{Itr.Iterate}^j(\mathsf{P}, v_0, (m_1, \dots, m_j))$ for all $1 \le j \le k$. Then, $\mathcal{I} = (\mathsf{Itr.Setup}, \mathsf{Itr.Setup\text{-}Enforce}, \mathsf{Itr.Iterate})$ is said to be *enforcing* if

$$v_k = \mathsf{Itr.Iterate}(\mathsf{P}, v', m') \implies (v', m') = (v_{k-1}, m_k).$$

Note that this is an information-theoretic property.

D Proof of Simulation Security for Succinct Partial Randomized Encodings

Proof of Theorem 8.1 : We will prove this theorem via a sequence of hybrid experiments. First, let us consider the top-level hybrid experiments.

Hyb_0 This hybrid corresponds to the real world security experiment. The challenger honestly chooses $(H_{\mathsf{aux}}, h_{\mathsf{aux}}) \leftarrow \mathsf{Preprocess}(1^\lambda, x_2)$, computes $\mathsf{Enc} \leftarrow \mathsf{Encode}(1^\lambda, M, T, x, (H_{\mathsf{aux}}, h_{\mathsf{aux}}))$. It sends Enc to $\mathcal{A}$ and $\mathcal{A}$ outputs its view α.

Hyb_1 Let $t^* = \min(T, \mathsf{Time}\,(M, (x, x_2)))$ and $b^* = \mathsf{TM}\,(M, (x, x_2), T)$. In this hybrid, the challenger outputs an obfuscation of $\mathsf{Prog\text{-}1}\{t^*, K_A, K_E, b^*\}$[17] (defined in Figure 17) as part of the encoding. This program is similar to Prog, however, for input $t > t^*$, it outputs $\perp$. At $t = t^*$, it outputs b^*.

Next, we define a sequence of hybrids $\mathsf{Hyb}_{2,i}$ and $\mathsf{Hyb}'_{2,i}$, where $1 \le i \le t^*$. Let erase be a symbol not present in Σ_{tape}.

$\mathsf{Hyb}_{2,i}$ In this hybrid, the challenger outputs an obfuscation of $\mathsf{Prog\text{-}2\text{-}}i\{i, t^*, K_E, K_A, b^*\}$ as part of the encoding. $\mathsf{Prog\text{-}2\text{-}}i$ also rejects on input $t > t^*$, and outputs b^* on t^* if the signature is the correct one. For $t < i$, its input output behavior is similar to that of Prog. However, for $i \le t < t^*$, on receiving a valid signature, it simply outputs encryptions of erase as the encryption of the state and symbol. It accumulates and iterates accordingly.

[17]Note that this program has other constants hardwired, which are dropped for simplicity of notations

Prog-1

Constants: Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_1, \mathsf{tmf}_2 \rangle$, time bound T, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A \in \mathcal{K}$, SSB Hash function H_{aux} and hash value h_{aux}, b^*, t^*.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value v_{in}, signature σ_{in}, accumulator proof π, SSB proof π_{aux}, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

```
1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. If $t > t^*$, output $\perp$.
3. **Verifications**
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
4. **Computing next state and symbol (encrypted)**
   (a) If $t = t^*$, output $b^*$.
   (b) Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
   (c) Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
   (d) Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
   (e) If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
   (f) Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
5. **Update accumulator, iterator and compute new signature**
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$ and $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
6. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathrm{out}}, \sigma_{\mathrm{out}}$.
```

Figure 17: Prog-1

$\mathsf{Hyb}'_{2,i}$ In this hybrid, the challenger runs M for $i-1$ steps and computes the state st^* and symbol sym^* written at $(i-1)^{th}$ step. Next, it computes $(r_{i-1,1}, r_{i-1,2}, r_{i-1,3}) = F(K_E, i-1)$, $\mathsf{sk} = \mathsf{SKE.Setup}(1^\lambda; r_{i-1,1})$, $\mathsf{ct}_1 = \mathsf{SKE.Enc}(\mathsf{sk}, \mathsf{sym}^*; r_{i-1,2})$ and $\mathsf{ct}_2 = \mathsf{SKE.Enc}(\mathsf{sk}, \mathsf{st}^*; r_{i-1,3})$. It then computes the obfuscation of $W_i = \mathsf{Prog}'\text{-}2\text{-}i\{i, t^*, K_E, K_A, \mathsf{ct}_1, \mathsf{ct}_2, b^*\}$ (defined in Figure 19), which has the ciphertexts ct_1 and ct_2 hardwired. On input corresponding to step $i-1$, W_i checks if the signature is valid, and if so, it outputs ct_1 and ct_2 without decrypting.

Hyb_3 This corresponds to the ideal world. Note that the obfuscated program in the hybrid experiment $\mathsf{Hyb}_{2,1}$ corresponds to the ideal world. The challenger, in this experiment, only requires $|M|$, tmf, $\mathsf{tmf}_{\mathsf{aux}}$, H_{aux}, $h_{\mathsf{aux}} = H_{\mathsf{aux}}(x_2)$, t^* and res to compute $\mathsf{Prog}\text{-}(1, a)$, and all these are given as inputs to the simulator.

Analysis Let $\mathsf{Adv}^x_{\mathcal{A}}$ denote the advantage of adversary $\mathcal{A}$ in hybrid Hyb_x, and $\mathsf{Adv}'^x_{\mathcal{A}}$ the advantage of $\mathcal{A}$ in Hyb'_x.

Lemma D.1. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF, Itr is an iterator satisfying Definitions C.9 and C.10, Acc is an accumulator satisfying Definitions C.5,

```
Prog-2-$i$

Constants: Index $i$, Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_1, \mathsf{tmf}_2 \rangle$, time bound $T$, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A \in \mathcal{K}$, SSB Hash function $H_{\mathsf{aux}}$ and hash value $h_{\mathsf{aux}}$, $b^*, t^*$.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value $v_{\mathsf{in}}$, signature $\sigma_{\mathrm{in}}$, accumulator proof $\pi$, SSB proof $\pi_{\mathsf{aux}}$, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. If $t > t^*$, output $\perp$.
3. Verifications
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
4. Computing next state and symbol (encrypted)
   (a) If $t = t^*$, output $b^*$.
   (b) If $i \leq t < t^*$, compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,3})$.
       Else do the following:
       i. Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
       ii. Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
       iii. Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
       iv. If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
       v. Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
5. Update accumulator, iterator and compute new signature
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$ and $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
6. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```

Figure 18: Prog-2-i

C.6, C.7 and C.8, $\mathcal{S}$ is a splittable signature scheme satisfying security Definitions C.1, C.2, C.3 and C.4 and SSB is a somewhere statistically binding scheme satisfying Definition B.1, for any PPT adversary $\mathcal{A}$, $\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^1_{\mathcal{A}} \leq \text{negl}(\lambda)$.

The proof of this lemma is contained in Section D.1.

Claim D.1. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, for any adversary $\mathcal{A}$, $\mathsf{Adv}^1_{\mathcal{A}} - \mathsf{Adv}^{2,t^*}_{\mathcal{A}} \leq \text{negl}(\lambda)$.

Proof. We note that the programs Prog-1 and Prog-2-t^* are functionally identical. ∎

Lemma D.2. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF, Itr is an iterator satisfying Definitions C.9 and C.10, Acc is an accumulator satisfying Definitions C.5, C.6, C.7 and C.8, $\mathcal{S}$ is a splittable signature scheme satisfying security Definitions C.1, C.2, C.3 and C.4,

```
$\mathsf{Prog}'$-2-$i$

Constants: Index $i$, Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_1, \mathsf{tmf}_2 \rangle$, time bound $T$, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A \in \mathcal{K}$, SSB Hash function $H_{\mathsf{aux}}$ and hash value $h_{\mathsf{aux}}$, $b^*, t^*$, $\mathsf{ct}_1$, $\mathsf{ct}_2$.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym,in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st,in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value $v_{\mathsf{in}}$, signature $\sigma_{\mathrm{in}}$, accumulator proof $\pi$, SSB proof $\pi_{\mathsf{aux}}$, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. If $t > t^*$, output $\perp$.
3. Verifications
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym,in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st,in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
4. Computing next state and symbol (encrypted)
   (a) If $t = t^*$, output $b^*$.
   (b) If $i \leq t < t^*$, compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym,out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st,out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,3})$.
       Else if $t = i - 1$, set $\mathsf{ct}_{\mathsf{sym,out}} = \mathsf{ct}_1$ and $\mathsf{ct}_{\mathsf{st,out}} = \mathsf{ct}_2$.
       Else do the following:
       i. Let $(r_{\ell\text{-w},1}, r_{\ell\text{-w},2}, r_{\ell\text{-w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-w}}, \mathsf{ct}_{\mathsf{sym,in}})$.
       ii. Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st,in}})$.
       iii. Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
       iv. If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
       v. Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym,out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st,out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
5. Update accumulator, iterator and compute new signature
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym,out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st,in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st,out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$ and $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
6. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym,out}}, \mathsf{ct}_{\mathsf{st,out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```

Figure 19: Prog'-2-i

and SSB is a somwhere statistically binding scheme satisfying Definition B.1, for any PPT adversary $\mathcal{A}$, $\mathsf{Adv}^{2,i}_{\mathcal{A}} - \mathsf{Adv}^{\prime 2,i}_{\mathcal{A}} \leq \mathrm{negl}(\lambda)$.

The proof of this lemma is contained in Section D.2.

Lemma D.3. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF, SKE is $\mathsf{IND\text{-}CPA}$ secure, for any adversary $\mathcal{A}$, $\mathsf{Adv}^{\prime 2,i}_{\mathcal{A}} - \mathsf{Adv}^{2,i-1}_{\mathcal{A}} \leq \mathrm{negl}(\lambda)$.

The proof of this lemma is contained in Section D.3.

Lemma D.4. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF, SKE is $\mathsf{IND\text{-}CPA}$ secure, for any adversary $\mathcal{A}$, $\mathsf{Adv}^{3}_{\mathcal{A}} - \mathsf{Adv}^{2,1}_{\mathcal{A}} \leq \mathrm{negl}(\lambda)$.

The proof of this lemma is contained in Section D.4.

D.1 Proof of Lemma D.1

Proof Outline We will first define hybrid experiments H_{int}, H'_{int} and H_{abort}.

Hybrid H_{int} In this hybrid, the challenger first receives $M, (x_1, x_2)$ from the adversary. It computes $t^* = \min(T, \mathsf{Time}(M, (x_1, x_2)))$, and chooses an SSB hash H_{aux} enforcing at $\mathsf{tmf}_{\mathsf{aux}}(t^*-2)$. Next, it computes $h_{\mathsf{aux}} = H_{\mathsf{aux}}(x_2)$ and sets $\mathsf{ek} = (H_{\mathsf{aux}}, h_{\mathsf{aux}})$.

It then computes the correct message m_{t^*-1} output by Prog at time t^*-1 (here, 'message' refers to the tuple signed by the program Prog. This consists of the iterator value, encryption of state, accumulator value and position on worktape where the new encrypted symbol is written). Next, it outputs an obfuscation of $P_{int} = P_{int}\{t^*, K_E, K_A, K_B, m_{t^*-1}\}$ (defined in Figure 20) which has m_{t^*-1} hardwired. It accepts only 'A' type signatures. However, at $t = t^*-1$, it checks if the outgoing message is m_{t^*-1}. If so, it outputs an 'A' type signature, else it outputs a 'B' type signature.

Program P_{int}

Constants: Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_{\mathrm{wk}}, \mathsf{tmf}_{\mathsf{aux}}\rangle$, time bound T, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A \in \mathcal{K}$, SSB Hash function H_{aux} and hash value h_{aux}, t^*, m_{t^*-1}.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value v_{in}, signature σ_{in}, accumulator proof π, SSB proof π_{aux}, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

```
1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. Verifications
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
3. Computing next state and symbol (encrypted)
   (a) Let $(r_{\ell\text{-w},1}, r_{\ell\text{-w},2}, r_{\ell\text{-w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
   (b) Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
   (c) Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
   (d) If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
   (e) Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$
       and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
4. Update accumulator, iterator and compute new signature
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$.
       If $t = t^*-1$ and $m_{\mathrm{out}} = m_{t^*-1}$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
       Else if $t = t^*-1$ and $m_{\mathrm{out}} \neq m_{t^*-1}$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_B, m_{\mathrm{out}})$.
       Else, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
5. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```

Figure 20: Program P_{int}

Hybrid H'_{int} This hybrid is similar to H_{int}, except that the challenger also computes $b^* = \mathsf{TM}(M, (x, x_2), T)$. The hash function H_{aux} is enforcing at t^*-1, and the challenger outputs an obfuscation of $P'_{int} = P'_{int}\{t^*, K_E, K_A, K_B, m_{t^*-1}, b^*\}$ (defined in Figure 21). This program is identical to P_{int}, except for inputs corresponding to $t = t^*$. At $t = t^*$, the program verifies the validity of signature, and then outputs b^* (which it has hardwired). It does not decrypt the ciphertexts and compute the final output.

```
Program $P'_{int}$

Constants: Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_{\mathrm{wk}}, \mathsf{tmf}_{\mathsf{aux}}\rangle$, time bound $T$, Public parameters
for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A \in \mathcal{K}$, SSB Hash
function $H_{\mathsf{aux}}$ and hash value $h_{\mathsf{aux}}$, $m_{t^*-1}$, $b^*$.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$,
encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value $v_{\mathrm{in}}$, signature $\sigma_{\mathrm{in}}$, accumulator proof
$\pi$, SSB proof $\pi_{\mathsf{aux}}$, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

  1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
  2. Verifications
     (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
     (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
     (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
     (d) Let $m_{\mathrm{in}} = (v_{\mathrm{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
  3. Computing next state and symbol (encrypted)
     (a) If $t = t^*$, output $b^*$.
     (b) Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} =$
         $\mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
     (c) Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
     (d) Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
     (e) If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
     (f) Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$
         and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
  4. Update accumulator, iterator and compute new signature
     (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
     (b) Compute $v_{\mathrm{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
     (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
     (d) Let $m_{\mathrm{out}} = (v_{\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$.
         If $t = t^* - 1$ and $m_{\mathrm{out}} = m_{t^*-1}$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
         Else if $t = t^* - 1$ and $m_{\mathrm{out}} \neq m_{t^*-1}$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_B, m_{\mathrm{out}})$.
         Else, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
  5. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathrm{out}}, \sigma_{\mathrm{out}}$.
```

Figure 21: Program P'_{int}

Hybrid H_{abort} In this hybrid, the challenger outputs an obfuscation of $P_{abort}\{t^*, K_A, K_E, b^*\}$ (defined in Figure 22). This program is similar to P'_{int}, except that it does not output 'B' type signatures.

Let $\mathsf{Adv}^{int}_{\mathcal{A}}$, $\mathsf{Adv}'^{int}_{\mathcal{A}}$, $\mathsf{Adv}^{abort}_{\mathcal{A}}$ be the advantages of an adversary $\mathcal{A}$ in H_{int}, H'_{int} and H_{abort} respectively. Recall $\mathsf{Adv}^0_{\mathcal{A}}$ and $\mathsf{Adv}^1_{\mathcal{A}}$ denote $\mathcal{A}$'s advantage in Hyb_0 and Hyb_1 respectively.

Lemma D.5. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF, Itr is an iterator satisfying Definitions C.9 and C.10, Acc is an accumulator satisfying Definitions C.5, C.6, C.7 and C.8, $\mathcal{S}$ is a splittable signature scheme satisfying security Definitions C.1, C.2, C.3 and C.4, SSB is a somewhere statistically binding hash function satisfying Definition B.1, $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^{int}_{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

Proof. The proof of this lemma is very similar to the corresponding proof in [KLW14] (Lemma B.1), except

Program P_{abort}

Constants: Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_{\mathrm{wk}}, \mathsf{tmf}_{\mathsf{aux}}\rangle$, time bound T, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A \in \mathcal{K}$, SSB Hash function H_{aux} and hash value h_{aux}, m_{t^*-1}, b^*.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value v_{in}, signature σ_{in}, accumulator proof π, SSB proof π_{aux}, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

```
1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. Verifications
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
3. Computing next state and symbol (encrypted)
   (a) If $t = t^*$, output $b^*$.
   (b) Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
   (c) Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
   (d) Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
   (e) If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
   (f) Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
4. Update accumulator, iterator and compute new signature
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$ and $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
5. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```

Figure 22: Program P_{abort}

for the SSB part. Therefore, in this section, we will give an outline of the proof, consisting of the outer hybrids, and refer to appropriate claims in [KLW14].

We will first define intermediate hybrids H_0, H_1 and $H_{2,j,0}, H_{2,j,1}$ and $H_{2,j,2}$ for $0 \leq j < t^*$.

Hybrid H_0 The challenger outputs $P_0 = \mathsf{Prog}\{t^*, K_E, K_A\}$.

Hybrid H_1 The challenger outputs $P_1 = P_1\{t^*, K_E, K_A, K_B\}$ (defined in Figure 23). This is similar to $\mathsf{Prog\text{-}1}$ defined in Figure 17. This program has PRF key K_B hardwired and accepts both 'A' and 'B' type signatures for $t < t^*$. If the incoming signature is of type α, then so is the outgoing signature.

Next, we define $3t^*$ intermediate hybrid experiments $H_{2,j,0}, H_{2,j,1}, H_{2,j,2}$ for $1 \leq j \leq t^* - 1$.

Hybrid $H_{2,j,0}$ In this hybrid, the challenger sets the SSB hash function to be binding at the aux-tape position read by the Turing machine at step j. Given machine M and input (x_1, x_2), the challenger first computes $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(j-1)$, and then uses $\mathsf{pos}_{\mathsf{aux}}$ to sample $H_{\mathsf{aux}} \leftarrow \mathsf{SSB.Gen}(1^\lambda, 1^{|x_2|}, \mathsf{pos}_{\mathsf{aux}})$. It sets $h_{\mathsf{aux}} = H_{\mathsf{aux}}(x_2)$. The remaining experiment is identical to $H_{2,j-1,2}$ (if $j = 1$, then the remaining experiment is same as H_1).

P_1

Constants: Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_{\mathrm{wk}}, \mathsf{tmf}_{\mathsf{aux}}\rangle$, time bound T, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A, K_B \in \mathcal{K}$, SSB Hash function H_{aux} and hash value h_{aux}.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value v_{in}, signature σ_{in}, accumulator proof π, SSB proof π_{aux}, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. **Verifications**
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $F(K_B, t-1) = r_{S,B}$. Compute $(\mathrm{SK}_B, \mathrm{VK}_B, \mathrm{VK}_{B,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,B})$.
   (e) Let $F(K_B, t) = r'_{S,B}$. Compute $(\mathrm{SK}'_B, \mathrm{VK}'_B, \mathrm{VK}'_{B,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,B})$.
   (f) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{*,\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$ and $\alpha =$'-'.
   If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 1$ set $\alpha =$'A'.
   If $\alpha =$'-' **and** $t \geq t^*$ output $\perp$.
   If $\alpha =$ '-' **and** $\mathsf{Spl.Verify}(\mathrm{VK}_B, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 1$ set $\alpha =$'B'.
   If $\alpha =$ '-' output $\perp$.
3. **Computing next state and symbol (encrypted)**
   (a) Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
   (b) Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
   (c) Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
   (d) If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
   (e) Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
4. **Update accumulator, iterator and compute new signature**
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{*,\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$ and $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_\alpha, m_{\mathrm{out}})$.
5. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.

Figure 23: P_1

Hybrid $H_{2,j,1}$ In this hybrid, the challenger outputs an obfuscation of $P_{2,j} = P_{2,j}\{j, t^*, K_E, K_A, K_B, m_j\}$. This circuit, defined in Figure 24, accepts 'B' type signatures only for inputs corresponding to $j+1 \leq t \leq t^*-1$. It also has the correct output message for step j - m_j hardwired. If an input has $j+1 \leq t \leq t^*-1$, then the output signature, if any, is of the same type as the incoming signature. If $t = j$, the program outputs an 'A' type signature if $m_{\mathrm{out}} = m_j$, else it outputs a 'B' type signature.

Hybrid $H_{2,j,2}$ In this hybrid, the challenger outputs an obfuscation of $P'_{2,j} = P'_{2,j}\{j, t^*, K_E, K_A, K_B, m_j\}$. This circuit, defined in Figure 25, accepts 'B' type signatures only for inputs corresponding to $j+2 \leq t \leq t^*-1$. It also has the correct input message m_j for step $j+1$ hardwired. If $t = j+1$ and $m_{\mathrm{in}} = m_j$ it outputs an 'A' type signature, else it outputs a 'B' type signature. If an input has $j+2 \leq t \leq t^*-1$, then the output signature, if any, is of the same type as the incoming signature.

Analysis

$P_{2,j}$

Constants: Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_{\mathrm{wk}}, \mathsf{tmf}_{\mathsf{aux}}\rangle$, time bound T, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A, K_B \in \mathcal{K}$, SSB Hash function H_{aux} and hash value h_{aux}, index j, message m_j.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value v_{in}, signature σ_{in}, accumulator proof π, SSB proof π_{aux}, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

```
1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. **Verifications**
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $F(K_B, t-1) = r_{S,B}$. Compute $(\mathrm{SK}_B, \mathrm{VK}_B, \mathrm{VK}_{B,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,B})$.
   (e) Let $F(K_B, t) = r'_{S,B}$. Compute $(\mathrm{SK}'_B, \mathrm{VK}'_B, \mathrm{VK}'_{B,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,B})$.
   (f) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{*,\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$ and $\alpha =$'-'.
       If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 1$ set $\alpha =$'A'.
       If $\alpha =$'-' **and** $(t \geq t^*$ **or** $t \leq j)$ output $\perp$.
       If $\alpha =$ '-' **and** $\mathsf{Spl.Verify}(\mathrm{VK}_B, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 1$ set $\alpha =$'B'.
       If $\alpha =$ '-' output $\perp$.
3. **Computing next state and symbol (encrypted)**
   (a) Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
   (b) Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
   (c) Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
   (d) If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
   (e) Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$
       and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
4. **Update accumulator, iterator and compute new signature**
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = Reject$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) If $t = j$ **and** $m_{\mathrm{out}} = m_j$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
       Else if $t = j$ **and** $m_{\mathrm{out}} \neq m_j$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_B, m_{\mathrm{out}})$.
       Else $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_\alpha, m_{\mathrm{out}})$.
5. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```

Figure 24: $P_{2,j}$

Claim D.2. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a secure puncturable PRF and $\mathcal{S}$ is a splittable signature scheme satisfying Definition C.1, for any PPT adversary $\mathcal{A}$, $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^1_{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

The proof of this claim is similar to the proof of Claim B.1 in [KLW14].

Claim D.3. Let $0 \leq j \leq t^* - 2$. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, Itr is an iterator satisfying indistinguishability of Setup (Definition C.9) and is enforcing (Definition C.10), Acc is an accumulator satisfying indistinguishability of Read/Write Setup (Definitions C.5 and C.6) and is Read/Write enforcing (Definitions C.7 and C.8) and SSB satisfies Definition B.1, for any PPT adversary $\mathcal{A}$, $|\mathsf{Adv}^{2,j,0}_{\mathcal{A}} - \mathsf{Adv}^{2,j,1}_{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

Proof. This proof requires a sequence of sub-sub-hybrids, similar to the proof of Claim B.4 (which is similar

$P'_{2,j}$

Constants: Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_{\mathrm{wk}}, \mathsf{tmf}_{\mathsf{aux}} \rangle$, time bound T, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A, K_B \in \mathcal{K}$, SSB Hash function H_{aux} and hash value h_{aux}, index j, message m_j.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value v_{in}, signature σ_{in}, accumulator proof π, SSB proof π_{aux}, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. **Verifications**
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $F(K_B, t-1) = r_{S,B}$. Compute $(\mathrm{SK}_B, \mathrm{VK}_B, \mathrm{VK}_{B,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,B})$.
   (e) Let $F(K_B, t) = r'_{S,B}$. Compute $(\mathrm{SK}'_B, \mathrm{VK}'_B, \mathrm{VK}'_{B,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,B})$.
   (f) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{*,\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$ and $\alpha =$'-'.
   If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 1$ set $\alpha =$'A'.
   If $\alpha =$'-' **and** $(t \geq t^*$ **or** $t \leq j+1)$ output $\perp$.
   If $\alpha =$ '-' **and** $\mathsf{Spl.Verify}(\mathrm{VK}_B, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 1$ set $\alpha =$'B'.
   If $\alpha =$ '-' output $\perp$.
3. **Computing next state and symbol (encrypted)**
   (a) Let $(r_{\ell\text{-w},1}, r_{\ell\text{-w},2}, r_{\ell\text{-w},3}) = F(K_E, \ell\text{-w})$, $\mathsf{sk}_{\ell\text{-w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
   (b) Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
   (c) Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
   (d) If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
   (e) Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
4. **Update accumulator, iterator and compute new signature**
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) If $t = j+1$ **and** $m_{\mathrm{in}} = m_j$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
   Else if $t = j+1$ **and** $m_{\mathrm{in}} \neq m_j$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_B, m_{\mathrm{out}})$.
   Else $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_\alpha, m_{\mathrm{out}})$.
5. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.

Figure 25: $P'_{2,j}$

to the proof of Lemma 6.3, described in Section A.3) in [KLW14]. However, we also require the SSB enforcing property here. Therefore, we will describe the sub-sub-hybrids at a high level here.

Hybrid h_1: In this hybrid, the challenger uses 'read enforced' setup for the accumulator. The challenger computes the first $\ell_{\mathsf{inp}} + j - 1$ 'correct tuples' for the accumulator. Let $\mathsf{enf} = ((\mathsf{inp}[0], 0), \ldots, (\mathsf{inp}[\ell_{\mathsf{inp}} - 1], \ell_{\mathsf{inp}} - 1), (\mathsf{ct}_{\mathsf{sym},w,1}, \mathsf{pos}_0), \ldots, (\mathsf{ct}_{\mathsf{sym},w,j}, \mathsf{pos}_{j-1}))$. The challenger computes $(\mathsf{P}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{\mathrm{STORE}_0}) \leftarrow \mathsf{Acc.Setup\text{-}Enforce\text{-}Read}(1^\lambda, T, \mathsf{enf}, \mathsf{pos}_{j-1})$. The remaining steps are same as in the previous hybrid.

This hybrid is indistinguishable from $H_{2,j,0}$ because the accumulator satisfies Definition C.5.

Hybrid h_2 In this hybrid, the challenger uses program P_2, which is similar to $P'_{2,j-1}$. However, in addition to checking if $m_{\mathrm{in}} = m_{j-1}$, it also checks if $(v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}) = (v_j, \mathsf{ct}_{\mathsf{st},j}, \mathsf{ct}_{\mathsf{sym},j})$.

Hybrids h_1 and h_2 are indistinguishable because the programs P_2 and $P'_{2,j}$ are functionally identical. Here, we use the fact that since the accumulator and SSB are read-enforcing, if $m_{\text{in}} = m_{j-1}$, then $(\mathsf{sym}_{\text{out}}, \mathsf{st}_{\text{out}}) = (\mathsf{sym}_j, \mathsf{st}_j)$.

Hybrid h_3 In this hybrid, the challenger uses normal setup instead of read-enforced setup.

Since the accumulator satisfies Definition C.5, h_2 and h_3 are computationally indistinguishable.

Hybrid h_4 In this hybrid, the challenger 'write enforces' the accumulator. As in hybrid H_1, the challenger computes the first $\ell_{\mathsf{inp}} + j$ 'correct tuples' to be accumulated. Let $\mathsf{sym}_{w,k}, \mathsf{pos}_k$ be the symbol output and the position after the k^{th} step. The challenger computes $(\mathsf{P}_{\mathsf{Acc}}, \widetilde{w}_0, \widetilde{store}_0) \leftarrow \mathsf{Acc.Setup\text{-}Enforce\text{-}Write}(1^\lambda, T, \mathsf{enf})$, where $\mathsf{enf} = ((\mathsf{inp}[0], 0), \ldots, (\mathsf{inp}[\ell_{\mathsf{inp}} - 1], \ell_{\mathsf{inp}} - 1), (\mathsf{ct}_{\mathsf{sym},w,1}, \mathsf{pos}_0), \ldots, (\mathsf{ct}_{\mathsf{sym},w,j}, \mathsf{pos}_{j-1}), (\mathsf{ct}_{\mathsf{sym},w,j+1}, \mathsf{pos}_j))$. The remaining computation is same as in previous step.

Hybrids h_3 and h_4 are computationally indistinguishable because accumulator satisfies Definition C.6.

Hybrid h_5 In this experiment, the challenger outputs an obfuscation of P_5, which is similar to P_2. However, on input where $t = j$, before computing signature, it also checks if $w_{\text{out}} = w_{j+1}$. Therefore, it checks whether $m_{\text{in}} = m_{j-1}$ and $m_{\text{out}} = m_j$.

Hybrids h_4 and h_5 are computationally indistinguishable because the programs P_2 and P_5 are functionally identical. If $m_{\text{in}} = m_{j-1}$ and $(v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\text{out}}, \mathsf{ct}_{\mathsf{sym},\text{out}}) = (v_j, \mathsf{ct}_{\mathsf{st},j}, \mathsf{ct}_{\mathsf{sym},j})$, then $w_{\text{in}} = w_{j-1}$. Therefore, using the write-enforcing property, we get that $w_{\text{out}} = w_j$.

Hybrid h_6 This experiment is similar to the previous one, except that the challenger uses normal setup for accumulator instead of 'enforcing write'.

Hybrids h_5 and h_6 are computationally indistinguishable because accumulator satisfies Definition C.6.

Hybrid h_7 This experiment is similar to the previous one, except that the challenger uses enforced setup for iterator instead of normal setup. It first computes $\mathsf{P}_{\mathsf{Acc}}, w_0, \text{STORE}_0$ as in the previous hybrid. Next, it computes the first j 'correct messages' for the iterator. Let $\mathsf{enf} = ((\mathsf{ct}_{\mathsf{st},0}, w_0, \mathsf{pos}_0), \ldots, (\mathsf{ct}_{\mathsf{st},j-1}, w_{j-1}, \mathsf{pos}_{j-1}))$. It computes $(\mathsf{P}_{\mathsf{Itr}}, v_0) \leftarrow \mathsf{Itr.Setup\text{-}Enforce}(1^\lambda, T, \mathsf{enf})$. The remaining hybrid proceeds as the previous one.

Hybrids h_6 and h_7 are indistinguishable because of iterator's indistinguishability of setup.

Hybrid h_8 In this experiment, the challenger outputs an obfuscation of P_8, which is similar to P_5, except that it only checks if $m_{\text{out}} = m_j$.

Hybrids h_7 and h_8 are indistinguishable because the programs P_5 and P_8 are functionally identical (the argument is identical to the proof of Claim A.28 in [KLW14]).

Hybrid h_9 This experiment is identical to $H_{2,j,1}$. It is indistinguishable from h_8 because of iterator's setup indistinguishability property.

■

Claim D.4. Let $0 \leq j \leq t^* - 1$. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF and $\mathcal{S}$ is a splittable signature scheme satisfying definitions C.1, C.2, C.3 and C.4, for any PPT adversary $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{2,j,1} - \mathsf{Adv}_{\mathcal{A}}^{2,j,2}| \leq \text{negl}(\lambda)$.

The proof of this claim is similar to the proof of Claim B.3 in [KLW14].

Claim D.5. Assuming SSB satisfies Definition B.1, for any PPT adversary $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{2,j,2} - \mathsf{Adv}_{\mathcal{A}}^{2,j+1,0}| \leq \text{negl}(\lambda)$.

Proof. Note that the only difference between these two hybrid experiments is the index used for SSB setup. In $H_{2,j,2}$, the SSB is enforcing at $\mathsf{tmf}_{\mathsf{aux}}(j-1)$, while in $H_{2,j+1,0}$, it is enforcing at $\mathsf{tmf}_{\mathsf{aux}}(j)$. Using the index hiding property, we can argue that these two hybrids are computationally indistinguishable. ■

■

Lemma D.6. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF, Itr is an iterator satisfying Definitions C.9 and C.10, Acc is an accumulator satisfying Definitions C.5, C.6, C.7 and C.8, $\mathcal{S}$ is a splittable signature scheme satisfying security Definitions C.1, C.2, C.3 and C.4 and SSB satisfies Definition B.1, $|\mathsf{Adv}_{\mathcal{A}}^{int} - \mathsf{Adv}_{\mathcal{A}}^{'int}| \leq \text{negl}(\lambda)$.

Proof. To prove this lemma, we will define a sequence of hybrid experiments and show that they are computationally indistinguishable.

Hybrid H_0 In this experiment, the challenger outputs an obfuscation of $P_0 = P_{int}\{t^*, K_E, K_A, K_B, m_{t^*-1}\}$.

Hybrid H_1 In this hybrid, the challenger first computes the constants for program P_1 as follows:

1. PRF keys K_A and K_B are punctured at t^*-1 to obtain $K_A\{t^*-1\} \leftarrow F.\mathsf{Puncture}(K_A, t^*-1)$ and $K_B\{t^*-1\} \leftarrow F.\mathsf{Puncture}(K_B, t^*-1)$.
2. Let $r_c = F(K_A, t^*-1)$, $(\text{SK}_C, \text{VK}_C, \text{VK}_{C,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_C)$, $r_D = F(K_B, t^*-1)$, $(\text{SK}_D, \text{VK}_D, \text{VK}_{D,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_D)$.

It then outputs an obfuscation of $P_1 = P_1\{t^*, K_E, K_A\{t^*-1\}, K_B\{t^*-1\}, \text{VK}_C, \text{SK}_C, \text{SK}_D, m_{t^*-1}\}$ (defined in 26). P_1 is identical to P_0 on inputs corresponding to $t \neq t^*-1, t^*$. For $t = t^*-1$, it uses the hardwired signing keys. For $t = t^*$, it uses the hardwired verification key.

Hybrid H_2 In this hybrid, r_C and r_D are chosen uniformly at random; that is, the challenger computes $(\text{SK}_C, \text{VK}_C) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$ and $(\text{SK}_D, \text{VK}_D) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$.

Hybrid H_3 In this hybrid, the challenger computes constrained secret/verification keys. It computes splittable signature keys $(\sigma_{C,\mathsf{one}}, \text{VK}_{C,\mathsf{one}}, \text{SK}_{C,\text{abo}}, \text{VK}_{C,\text{abo}}) \leftarrow \mathsf{Spl.Split}(\text{SK}_C, m_{t^*-1})$ and $(\sigma_{D,\mathsf{one}}, \text{VK}_{D,\mathsf{one}}, \text{SK}_{D,\text{abo}}, \text{VK}_{D,\text{abo}}) \leftarrow \mathsf{Spl.Split}(\text{SK}_D, m_{t^*-1})$. It then outputs an obfuscation of $P_3 = P_1\{t^*, K_E, K_A\{t^*-1\}, K_B\{t^*-1\}, \text{VK}_{C,\mathsf{one}}, \sigma_{C,\mathsf{one}}, \text{SK}_{D,\text{abo}}, m_{t^*-1}\}$. Note that this program is identical to P_1, except that $\text{VK}_{C,\mathsf{one}}, \sigma_{C,\mathsf{one}}$ and $\text{SK}_{D,\text{abo}}$ are used instead of VK_C, SK_C and VK_D, and SK_C, VK_C, SK_D, VK_D are not hardwired in this program.

Hybrid H_4 In this hybrid, the challenger chooses $\mathsf{P}_{\mathsf{Acc}}, w_0, \text{STORE}_0$ using $\mathsf{Acc.Setup\text{-}Enforce\text{-}Read}$. It enforces the accumulator at position $\mathsf{tmf}(t^*-1)$, and then uses $\mathsf{P}_{\mathsf{Acc}}, w_0, \text{STORE}_0$, and proceeds as in previous experiment.

Hybrid H_5 In this hybrid, the challenger enforces the SSB hash function at position $\mathsf{tmf}_{\mathsf{aux}}(t^*-1)$.

Hybrid H_6 In this hybrid, the challenger first computes $b^* = M_b(x)$. It then outputs an obfuscation of $P_6 = P_6\{t^*, \mathsf{P}_{\mathsf{Acc}}, K_E, K_A\{t^*-1\}, K_B\{t^*-1\}, \text{VK}_{C,\mathsf{one}}, \sigma_{C,\mathsf{one}}, \text{SK}_{D,\text{abo}}, m_{t^*-1}, b^*\}$ (defined in Figure 27). This program differs from P_1 for inputs corresponding to $t = t^*$. Instead of decrypting, computing the next state and then encrypting, the program uses the hardwired output b^*.

Hybrid H_7 In this experiment, the challenger uses normal setup for Acc (that is, $\mathsf{Acc.Setup}$) instead of $\mathsf{Acc.Setup\text{-}Enforce\text{-}Read}$.

P_1

Constants: Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_{\mathrm{wk}}, \mathsf{tmf}_{\mathsf{aux}}\rangle$, time bound T, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A, K_B \in \mathcal{K}$, SSB Hash function H_{aux} and hash value h_{aux}, t^*, m_{t^*-1}, VK_C, SK_C, SK_D.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value v_{in}, signature σ_{in}, accumulator proof π, SSB proof π_{aux}, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

```
1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. Verifications
    (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
    (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
    (c) If $t \neq t^*$, let $r_{S,A} = F(K_A\{t^*-1\}, t-1)$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
        Else $\mathrm{VK}_A = \mathrm{VK}_C$.
    (d) If $t \neq t^*-1$, let $r'_{S,A} = F(K_A\{t^*-1\}, t)$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
    (e) If $t \neq t^*-1$, $r'_{S,B} = F(K_B\{t^*-1\}, t)$. Compute $(\mathrm{SK}'_B, \mathrm{VK}'_B, \mathrm{VK}'_{B,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,B})$.
    (f) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{*,\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
3. Computing next state and symbol (encrypted)
    (a) Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
    (b) Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
    (c) Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
    (d) If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
    (e) Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$
        and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
4. Update accumulator, iterator and compute new signature
    (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = Reject$, output $\perp$.
    (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
    (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
    (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$.
    (e) If $t = t^*-1$ and $m_{\mathrm{out}} = m_{t^*-1}$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}_C, m_{\mathrm{out}})$.
        Else if $t = t^*-1$ and $m_{\mathrm{out}} \neq m_{t^*-1}$ $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}_D, m_{\mathrm{out}})$.
        Else $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
5. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```

Figure 26: P_1

Hybrid H_8 This hybrid is identical to H'_{int}. In this experiment, the challenger outputs an obfuscation of P'_{int}.

Analysis Let $\mathsf{Adv}^x_{\mathcal{A}}$ denote the advantage of adversary $\mathcal{A}$ in hybrid H_x.

Claim D.6. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, for any PPT $\mathcal{A}$, $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^1_{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

Proof. The only difference between P_0 and P_1 is that P_0 uses puncturable PRF keys K_A, K_B, while P_1 uses keys $K_A\{t^*-1\}, K_B\{t^*-1\}$ punctured at t^*-1. It also has the secret key/verification key pair $(\mathrm{SK}_C, \mathrm{VK}_C)$ hardwired, which is computed using $F(K_A, t^*-1)$ and the secret key (SK_D) computed using $F(K_B, t^*-1)$. From the correctness of puncturable PRFs, it follows that the two programs have identical functionality, and therefore their obfuscations are computationally indistinguishable. ■

P_6

Constants: Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_{\mathrm{wk}}, \mathsf{tmf}_{\mathsf{aux}}\rangle$, time bound T, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A, K_B \in \mathcal{K}$, SSB Hash function H_{aux} and hash value h_{aux}, t^*, m_{t^*-1}, VK_C, SK_C, SK_D, hardwired output b^*.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value v_{in}, signature σ_{in}, accumulator proof π, SSB proof π_{aux}, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. **Verifications**
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) If $t \neq t^*$, let $r_{S,A} = F(K_A\{t^*-1\}, t-1)$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$. Else $\mathrm{VK}_A = \mathrm{VK}_C$.
   (d) If $t \neq t^*-1$, let $r'_{S,A} = F(K_A\{t^*-1\}, t)$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (e) If $t \neq t^*-1$, $r'_{S,B} = F(K_B\{t^*-1\}, t)$. Compute $(\mathrm{SK}'_B, \mathrm{VK}'_B, \mathrm{VK}'_{B,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,B})$.
   (f) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{*,\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
3. **Computing next state and symbol (encrypted)**
   (a) If $t = t^*$, output b^*.
   (b) Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
   (c) Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
   (d) Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
   (e) If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
   (f) Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
4. **Update accumulator, iterator and compute new signature**
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$.
   (e) If $t = t^*-1$ and $m_{\mathrm{out}} = m_{t^*-1}$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}_C, m_{\mathrm{out}})$. Else if $t = t^*-1$ and $m_{\mathrm{out}} \neq m_{t^*-1}$ $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}_D, m_{\mathrm{out}})$. Else $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
5. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.

Figure 27: P_6

Claim D.7. Assuming F is a selectively secure puncturable PRF, for any PPT $\mathcal{A}$, $|\mathsf{Adv}^1_{\mathcal{A}} - \mathsf{Adv}^2_{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

Proof. The proof of this claim follows from the selective security of puncturable PRF F. ∎

Claim D.8. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator and $\mathcal{S}$ satisfies $\mathrm{VK}_{\mathsf{one}}$ indistinguishability (Definition C.2), for any PPT $\mathcal{A}$, $|\mathsf{Adv}^2_{\mathcal{A}} - \mathsf{Adv}^3_{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

Proof. In order to prove this claim, we consider an intermediate hybrid program in which only the constrained secret keys $\sigma_{C,\mathsf{one}}$ and $\mathrm{SK}_{D,\mathrm{abo}}$ are hardwired, while VK_C is hardwired as the verification key. Using the security of $i\mathcal{O}$, we can argue that the intermediate step and H_2 are computationally indistinguishable. Next, we use $\mathrm{VK}_{\mathsf{one}}$ indistinguishability to show that the intermediate step and H_3 are computationally indistinguishable. ∎

Claim D.9. Assuming Acc satisfies indistinguishability of Read Setup (Definition C.5), for any PPT $\mathcal{A}$, $|\mathsf{Adv}^3_{\mathcal{A}} - \mathsf{Adv}^4_{\mathcal{A}}| \leq \text{negl}(\lambda)$.

Proof. The proof of this claim follows from Read Setup indistinguishability (Definition C.5). ■

Claim D.10. Assuming SSB satisfies Definition B.1, for any PPT $\mathcal{A}$, $|\mathsf{Adv}^4_{\mathcal{A}} - \mathsf{Adv}^5_{\mathcal{A}}| \leq \text{negl}(\lambda)$.

Proof. The proof of this claim follows from the index-hiding security of SSB. Note that the only difference between H_4 and H_5 is the index to which the SSB hash function is binding. In H_4, the SSB hash is binding to position $\mathsf{tmf}_{\mathsf{aux}}(t^* - 2)$ (note that it is binding to position $\mathsf{tmf}_{\mathsf{aux}}(t^* - 2)$ in H_{int}, and the SSB setup does not change from H_{int} to H_4). In H_5, it is binding to $\mathsf{tmf}_{\mathsf{aux}}(t^* - 1)$. Using the index-hiding security of SSB, it follows that these two hybrids are computationally indistinguishable. ■

Claim D.11. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, for any PPT $\mathcal{A}$, $|\mathsf{Adv}^5_{\mathcal{A}} - \mathsf{Adv}^6_{\mathcal{A}}| \leq \text{negl}(\lambda)$.

Proof. Note that $\mathsf{P}_{\mathsf{Acc}}$ and SSB are appropriately enforced, and $\text{VK}_{C,\mathsf{one}}$ accepts only signatures for m_{t^*-1}. As a result, if $\text{VK}_{C,\mathsf{one}}$ accepts the signature, then $m_{\text{in}} = m_{t^*-1}$ and hence $\mathsf{ct}_{\mathsf{st},\text{in}} = \mathsf{ct}_{\mathsf{st},t^*-1}$. Next, since $PPAcc$ is read enforcing, $\mathsf{ct}_{\mathsf{sym},\text{in}} = \mathsf{ct}_{\mathsf{sym},t^*-1}$, and since SSB is enforcing at $t^* - 1$, $\mathsf{sym}_{\mathsf{aux}} = x_{2,\mathsf{tmf}_{\mathsf{aux}}(t^*-1)}$. Therefore, $^*_{\text{out}} =^*_{t^*}$, which implies that the output is b^*. ■

Claim D.12. Assuming Acc satisfies indistinguishability of Read Setup (Definition C.5), for any PPT $\mathcal{A}$, $|\mathsf{Adv}^6_{\mathcal{A}} - \mathsf{Adv}^7_{\mathcal{A}}| \leq \text{negl}(\lambda)$.

Proof. The proof of this claim follows from Read Setup indistinguishability (Definition C.5). ■

Claim D.13. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF and $\mathcal{S}$ satisfies VK_{one} indistinguishability (Definition C.2), for any PPT $\mathcal{A}$, $|\mathsf{Adv}^7_{\mathcal{A}} - \mathsf{Adv}^8_{\mathcal{A}}| \leq \text{negl}(\lambda)$.

This step is the reverse of the step from H_0 to H_3. Therefore, using similar intermediate hybrid experiments, a similar proof works here as well.

■

Lemma D.7. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF, Itr is an iterator satisfying Definitions C.9 and C.10, Acc is an accumulator satisfying Definitions C.5, C.6, C.7 and C.8, $\mathcal{S}$ is a splittable signature scheme satisfying security Definitions C.1, C.2, C.3 and C.4 and SSB satisfies Definition B.1, $|\mathsf{Adv}'^{int}_{\mathcal{A}} - \mathsf{Adv}^{abort}_{\mathcal{A}}| \leq \text{negl}(\lambda)$.

The proof of this lemma is almost identical to the proof of Lemma D.5.

Lemma D.8. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure PRF and $\mathcal{S}$ satisfies VK_{rej} indistinguishability (Definition C.1), for any PPT adversary $\mathcal{A}$, $|\mathsf{Adv}^{abort}_{\mathcal{A}} - \mathsf{Adv}^1_{\mathcal{A}}| \leq \text{negl}(\lambda)$.

This proof is similar to the proof of Lemma B.4 in [KLW14].

```
                                   $W_{int}$

$\textbf{Constants}$: Index $i$, Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_1, \mathsf{tmf}_2 \rangle$, time bound $T$, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A, K_B \in \mathcal{K}$, SSB Hash function $H_{\mathsf{aux}}$ and hash value $h_{\mathsf{aux}}$, $b^*, t^*$, $m_{i-2}$.

$\textbf{Input:}$ Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value $v_{\mathsf{in}}$, signature $\sigma_{\mathrm{in}}$, accumulator proof $\pi$, SSB proof $\pi_{\mathsf{aux}}$, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

  1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
  2. If $t > t^*$, output $\perp$.
  3. Verifications
     (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
     (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
     (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
     (d) Let $F(K_B, t-1) = r_{S,B}$. Compute $(\mathrm{SK}_B, \mathrm{VK}_B, \mathrm{VK}_{B,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,B})$.
     (e) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
  4. Computing next state and symbol (encrypted)
     (a) If $t = t^*$, output $b^*$.
     (b) If $i \leq t < t^*$, compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,3})$.
         Else do the following:
          i. Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
         ii. Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
        iii. Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
         iv. If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
          v. Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
  5. Update accumulator, iterator and compute new signature
     (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = Reject$, output $\perp$.
     (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
     (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
     (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$.
         If $t = i-2$ and $m_{\mathrm{out}} = m_{i-2}$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
         Else if $t = i-2$ and $m_{\mathrm{out}} = m_{i-2}$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_B, m_{\mathrm{out}})$.
         Else $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
  6. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```

Figure 28: W_{int}

D.2 Proof of Lemma D.2

Proof Outline We will first define intermediate programs $W_{int} = \mathsf{Prog}\text{-}2\text{-}i_{int}\{i,\ t^*,\ K_E,\ K_A,\ K_B,\ m_{i-2}\}$ (defined in Figure 28) and $W'_{int} = \mathsf{Prog}'\text{-}2\text{-}i_{int}\{i,\ t^*,\ K_E,\ K_A,\ K_B,\ m_{i-2}, \mathsf{ct}_1, \mathsf{ct}_2\}$ (defined in 29). Both the programs have the correct message for the $(i-2)^{th}$ step - m_{i-2} hardwired, and also have a PRF key K_B for 'B' type signatures. In addition, W'_{int} also has ciphertexts ct_1 and ct_2 hardwired. These are encryptions of the state and symbol output at $(i-1)^{th}$ step, computed as described in hybrid $\mathsf{Hyb}'_{2,i}$.

Let H_{int} be a hybrid experiment in which the challenger first enforces the SSB hash, and then outputs an obfuscation of W_{int}, along with other elements of the encoding. Similarly, let H'_{int} be the hybrid experiment

```
$W'_{int}$

Constants: Index $i$, Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_1, \mathsf{tmf}_2 \rangle$, time bound $T$, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A, K_B \in \mathcal{K}$, SSB Hash function $H_{\mathsf{aux}}$ and hash value $h_{\mathsf{aux}}$, $b^*, t^*$, $m_{i-2}$, ciphertexts $\mathsf{ct}_1, \mathsf{ct}_2$.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value $v_{\mathsf{in}}$, signature $\sigma_{\mathrm{in}}$, accumulator proof $\pi$, SSB proof $\pi_{\mathsf{aux}}$, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. If $t > t^*$, output $\perp$.
3. Verifications
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $F(K_B, t-1) = r_{S,B}$. Compute $(\mathrm{SK}_B, \mathrm{VK}_B, \mathrm{VK}_{B,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,B})$.
   (e) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
4. Computing next state and symbol (encrypted)
   (a) If $t = t^*$, output $b^*$.
   (b) If $i \leq t < t^*$, compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,3})$.
       Else if $t = i-1$, set $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{ct}_1$, $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{ct}_2$.
       Else do the following:
         i. Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
        ii. Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
       iii. Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
        iv. If $\mathsf{st}_{\mathsf{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathsf{out}} = q_{\mathrm{acc}}$ output 1.
         v. Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
5. Update accumulator, iterator and compute new signature
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = Reject$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$.
       If $t = i-2$ and $m_{\mathrm{out}} = m_{i-2}$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
       Else if $t = i-2$ and $m_{\mathrm{out}} = m_{i-2}$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_B, m_{\mathrm{out}})$.
       Else $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
6. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```

Figure 29: W'_{int}

in which the challenger enforces the SSB hash and then outputs W'_{int}. For any PPT adversary $\mathcal{A}$, let $\mathsf{Adv}^{2,i}_{\mathcal{A}}$, $\mathsf{Adv}^{int}_{\mathcal{A}}$, $\mathsf{Adv}'^{int}_{\mathcal{A}}$, $\mathsf{Adv}'^{2,i+1}_{\mathcal{A}}$ denote the advantage of $\mathcal{A}$ in $\mathsf{Hyb}_{2,i}$, H_{int}, H'_{int} and $\mathsf{Hyb}'_{2,i}$ respectively.

Lemma D.9. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF, Itr is an iterator satisfying Definitions C.9 and C.10, Acc is an accumulator satisfying Definitions C.5, C.6, C.7 and C.8, $\mathcal{S}$ is a splittable signature scheme satisfying security Definitions C.1, C.2, C.3 and C.4 and SSB satisfies Definition B.1, $|\mathsf{Adv}^{2,i}_{\mathcal{A}} - \mathsf{Adv}^{int}_{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

Proof. The proof of this lemma is along the same lines as the proof of Lemma D.5. We will define similar hybrid experiments here.

Hybrid H_0 The challenger outputs $P_0 = \textsf{Prog-2-}i\{i,\ t^*,\ K_E,\ K_A\}$.

Hybrid H_1 The challenger outputs $P_1 = P_1\{i,\ t^*,\ K_E,\ K_A, K_B\}$. This program has PRF key K_B hardwired and accepts both 'A' and 'B' type signatures for $t \le i-2$. If the incoming signature is of type α, then so is the outgoing signature. It is defined in Figure 30.

P_1

Constants: Index i, Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_1, \mathsf{tmf}_2\rangle$, time bound T, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A, K_B \in \mathcal{K}$, SSB Hash function H_{aux} and hash value h_{aux}, b^*, t^*.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value v_{in}, signature σ_{in}, accumulator proof π, SSB proof π_{aux}, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

```
1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. If $t > t^*$, output $\perp$.
3. **Verifications**
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \ge t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $F(K_B, t-1) = r_{S,B}$. Compute $(\mathrm{SK}_B, \mathrm{VK}_B, \mathrm{VK}_{B,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,B})$.
   (e) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$ and $\alpha = $ 'A'.
       If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ and $t \ge i-1$, output $\perp$.
       Else if $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ set $\alpha = $ 'B'.
       If $\alpha = $ 'B' and $\mathsf{Spl.Verify}(\mathrm{VK}_B, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
4. **Computing next state and symbol (encrypted)**
   (a) If $t = t^*$, output $b^*$.
   (b) If $i \le t < t^*$, compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,3})$.
       Else do the following:
       i. Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
       ii. Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
       iii. Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
       iv. If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
       v. Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
5. **Update accumulator, iterator and compute new signature**
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $F(K_B, t) = r'_{S,B}$. Compute $(\mathrm{SK}'_B, \mathrm{VK}'_B, \mathrm{VK}'_{B,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,B})$.
   (e) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$ and $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_\alpha, m_{\mathrm{out}})$.
6. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```

Figure 30: P_1

Next, we define $2(i-1)$ intermediate circuits - $P_{2,j}, P'_{2,j}$ for $0 \leq j \leq i-2$.

Hybrid $H_{2,j}$ In this hybrid, the challenger outputs an obfuscation of $P_{2,j} = P_{2,j}\{i, j, t^*, K_E, K_A, K_B, m_j\}$. This circuit, defined in Figure 31, accepts 'B' type signatures only for inputs corresponding to $j+1 \leq t \leq i-2$. It also has the correct output message for step j - m_j hardwired. If an input has $j+1 \leq t \leq i-2$, then the output signature, if any, is of the same type as the incoming signature.

$P_{2,j}$

Constants: Index i, Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_1, \mathsf{tmf}_2\rangle$, time bound T, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A, K_B \in \mathcal{K}$, SSB Hash function H_{aux} and hash value h_{aux}, b^*, t^*, tuple m_j.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value v_{in}, signature σ_{in}, accumulator proof π, SSB proof π_{aux}, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. If $t > t^*$, output $\perp$.
3. **Verifications**
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $F(K_B, t-1) = r_{S,B}$. Compute $(\mathrm{SK}_B, \mathrm{VK}_B, \mathrm{VK}_{B,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,B})$.
   (e) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$ and $\alpha = $ 'A'.
   If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ and $(t \leq j$ or $t \geq i-1)$, output $\perp$.
   Else if $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ set $\alpha = $ 'B'.
   If $\alpha = $ 'B' amd $\mathsf{Spl.Verify}(\mathrm{VK}_B, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
4. **Computing next state and symbol (encrypted)**
   (a) If $t = t^*$, output b^*.
   (b) If $i \leq t < t^*$, compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,3})$.
   Else do the following:
      i. Let $(r_{\ell\text{-w},1}, r_{\ell\text{-w},2}, r_{\ell\text{-w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
      ii. Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
      iii. Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
      iv. If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
      v. Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
5. **Update accumulator, iterator and compute new signature**
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = Reject$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $F(K_B, t) = r'_{S,B}$. Compute $(\mathrm{SK}'_B, \mathrm{VK}'_B, \mathrm{VK}'_{B,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,B})$.
   (e) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$.
   If $t = j$ **and** $m_{\mathrm{out}} = m_j$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
   Else if $t = j$ **and** $m_{\mathrm{out}} \neq m_j$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_B, m_{\mathrm{out}})$.
   Else $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_\alpha, m_{\mathrm{out}})$.
6. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.

Figure 31: $P_{2,j}$

Hybrid $H'_{2,j}$ In this hybrid, the challenger outputs an obfuscation of $P'_{2,j} = P'_{2,j}\{i, j, t^*, K_E, K_A, K_B, m_j\}$. This circuit, defined in Figure 32, accepts 'B' type signatures only for inputs corresponding to $j+2 \le t \le i-2$. It also has the correct input message for step $j+1$ - m_j hardwired. If $t = j+1$ and $m_{\text{in}} = m_j$ it outputs an 'A' type signature, else it outputs a 'B' type signature. If an input has $j+2 \le t \le i-2$, then the output signature, if any, is of the same type as the incoming signature.

$P'_{2,j}$

Constants: Index i, Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}, \mathsf{tmf}_1, \mathsf{tmf}_2 \rangle$, time bound T, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A, K_B \in \mathcal{K}$, SSB Hash function H_{aux} and hash value h_{aux}, b^*, t^*, tuple m_j.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\text{in}}, \ell\text{-w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\text{in}}$, accumulator value $w_{\text{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value v_{in}, signature σ_{in}, accumulator proof π, SSB proof π_{aux}, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

```
1. Let $\mathsf{pos}_{\text{in}} = \mathsf{tmf}_{\text{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\text{out}} = \mathsf{tmf}_{\text{wk}}(t)$.
2. If $t > t^*$, output $\perp$.
3. Verifications
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\text{in}}, (\mathsf{ct}_{\mathsf{sym},\text{in}}, \ell\text{-w}), \mathsf{pos}_{\text{in}}, \pi) = 0$ or $\ell\text{-w} \ge t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $F(K_B, t-1) = r_{S,B}$. Compute $(\mathrm{SK}_B, \mathrm{VK}_B, \mathrm{VK}_{B,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,B})$.
   (e) Let $m_{\text{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\text{in}}, w_{\text{in}}, \mathsf{pos}_{\text{in}})$ and $\alpha = `A'$.
       If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\text{in}}, \sigma_{\text{in}}) = 0$ and ($t \le j+1$ or $t \ge i-1$), output $\perp$.
       Else if $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\text{in}}, \sigma_{\text{in}}) = 0$ set $\alpha = `B'$.
       If $\alpha = `B'$ amd $\mathsf{Spl.Verify}(\mathrm{VK}_B, m_{\text{in}}, \sigma_{\text{in}}) = 0$ output $\perp$.
4. Computing next state and symbol (encrypted)
   (a) If $t = t^*$, output $b^*$.
   (b) If $i \le t < t^*$, compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\text{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\text{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,3})$.
       Else do the following:
        i. Let $(r_{\ell\text{-w},1}, r_{\ell\text{-w},2}, r_{\ell\text{-w},3}) = F(K_E, \ell\text{-w})$, $\mathsf{sk}_{\ell\text{-w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-w}}, \mathsf{ct}_{\mathsf{sym},\text{in}})$.
       ii. Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\text{in}})$.
      iii. Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
       iv. If $\mathsf{st}_{\text{out}} = q_{\text{rej}}$ output 0. Else if $\mathsf{st}_{\text{out}} = q_{\text{acc}}$ output 1.
        v. Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\text{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\text{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
5. Update accumulator, iterator and compute new signature
   (a) Compute $w_{\text{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\text{in}}, (\mathsf{ct}_{\mathsf{sym},\text{out}}, t), \mathsf{pos}_{\text{in}}, \mathsf{acc\text{-}aux})$. If $w_{\text{out}} = Reject$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\text{in}}, w_{\text{in}}, \mathsf{pos}_{\text{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $F(K_B, t) = r'_{S,B}$. Compute $(\mathrm{SK}'_B, \mathrm{VK}'_B, \mathrm{VK}'_{B,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,B})$.
   (e) Let $m_{\text{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\text{out}}, w_{\text{out}}, \mathsf{pos}_{\text{out}})$.
       If $t = j+1$ and $m_{\text{in}} = m_j$, $\sigma_{\text{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\text{out}})$.
       Else if $t = j+1$ and $m_{\text{in}} \ne m_j$, $\sigma_{\text{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_B, m_{\text{out}})$.
       Else $\sigma_{\text{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_\alpha, m_{\text{out}})$.
6. Output $\mathsf{pos}_{\text{in}}, \mathsf{ct}_{\mathsf{sym},\text{out}}, \mathsf{ct}_{\mathsf{st},\text{out}}, w_{\text{out}}, v_{\mathsf{out}}, \sigma_{\text{out}}$.
```

Figure 32: $P'_{2,j}$

Analysis Let $\mathsf{Adv}^x_{\mathcal{A}}$ denote the advantage of adversary $\mathcal{A}$ in hybrid H_x.

Claim D.14. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a xsecure puncturable PRF and $\mathcal{S}$ is a splittable signature scheme satisfying Definition C.1, $\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^1_{\mathcal{A}} \leq \text{negl}(\lambda)$.

Proof. The proof of this claim is similar to the proof of Claim D.2. ∎

Claim D.15. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, $\mathsf{Adv}^1_{\mathcal{A}} - \mathsf{Adv}^{2,0}_{\mathcal{A}} \leq \text{negl}(\lambda)$.

Proof. Note that P_1 and $P_{2,0}$ have identical functionality. ∎

Claim D.16. Let $0 \leq j \leq i-2$. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF and $\mathcal{S}$ is a splittable signature scheme satisfying definitions C.1, C.2, C.3 and C.4, $\mathsf{Adv}^{2,j}_{\mathcal{A}} - \mathsf{Adv}'^{2,j}_{\mathcal{A}} \leq \text{negl}(\lambda)$.

Proof. The proof of this claim is similar to the proof of Claim D.4. ∎

Claim D.17. Let $0 \leq j \leq i-3$. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, Itr is an iterator satisfying indistinguishability of Setup (Definition C.9) and is enforcing (Definition C.10), and Acc is an accumulator satisfying indistinguishability of Read/Write Setup (Definitions C.5 and C.6) and is Read/Write enforcing (Definitions C.7 and C.8) and SSB satisfies Definition B.1, $\mathsf{Adv}'^{2,j}_{\mathcal{A}} - \mathsf{Adv}^{2,j+1}_{\mathcal{A}} \leq \text{negl}(\lambda)$.

Proof. This transition is similar to the transition from $H_{2,j,2}$ to $H_{2,j+1,1}$, and therefore the proof of this claim is similar to the proof of Claim D.5 and Claim D.3. ∎

Claim D.18. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, $\mathsf{Adv}^{2,i-2}_{\mathcal{A}} - \mathsf{Adv}^{int}_{\mathcal{A}} \leq \text{negl}(\lambda)$.

Proof. Note that $P_{2,i-2}$ and W_{int} are functionally identical circuits, and the SSB hash is also enforcing at the same position in both hybrids. ∎

∎

Lemma D.10. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF, Itr is an iterator satisfying Definitions C.9 and C.10, Acc is an accumulator satisfying Definitions C.5, C.6, C.7 and C.8, $\mathcal{S}$ is a splittable signature scheme satisfying security Definitions C.1, C.2, C.3 and C.4 and SSB satisfies Definition B.1, $|\mathsf{Adv}^{int}_{\mathcal{A}} - \mathsf{Adv}'^{int}_{\mathcal{A}}| \leq \text{negl}(\lambda)$.

Proof. The proof of this lemma is similar to the proof of Lemma D.6. To prove this lemma, we will define a sequence of hybrid experiments and show that they are computationally indistinguishable.

Hybrid H_0 In this experiment, the challenger outputs an obfuscation of $P_0 = W_{int} = \mathsf{Prog}\text{-}\mathsf{2}\text{-}i\{i,\ t^*,\ K_E,\ K_A,\ K_B,\ m_{i-2}\}$.

Hybrid H_1 In this hybrid, the challenger first computes the constants for program P_1 as follows:

1. PRF keys K_A and K_B are punctured at $i-2$ to obtain $K_A\{i-2\} \leftarrow F.\mathsf{Puncture}(K_A, i-2)$ and $K_B\{i-2\} \leftarrow F.\mathsf{Puncture}(K_B, i-2)$.
2. Let $r_c = F(K_A, i{-}2)$, $(\text{SK}_C, \text{VK}_C, \text{VK}_{C,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_C)$, $r_D = F(K_B, i{-}2)$, $(\text{SK}_D, \text{VK}_D, \text{VK}_{D,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_D)$.

It then outputs an obfuscation of $P_1 = P_1\{i, t^*, K_E, K_A\{i{-}2\}, K_B\{i{-}2\}, \text{VK}_{C,\mathsf{one}}, \text{SK}_{C,\mathsf{one}}, \text{SK}_{D,\mathsf{abo}}, m_{i-2}\}$ (defined in 33). P_1 is identical to P_0 on inputs corresponding to $t \neq i-1, i-2$. However, for $i-2$, its output signature is computed using either SK_C or SK_D. For inputs corresponding to $t = i-1$, it uses VK_C for the verification.

```
                                   $P_1$

Constants: Index $i$, Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_1, \mathsf{tmf}_2 \rangle$, time bound $T$, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A, K_B \in \mathcal{K}$, SSB Hash function $H_{\mathsf{aux}}$ and hash value $h_{\mathsf{aux}}$, $b^*, t^*$, $m_{i-2}$, $\mathrm{VK}_C$, $\sigma_C$, $\mathrm{SK}_D$.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value $v_{\mathsf{in}}$, signature $\sigma_{\mathrm{in}}$, accumulator proof $\pi$, SSB proof $\pi_{\mathsf{aux}}$, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. If $t > t^*$, output $\perp$.
3. Verifications
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) If $t \neq i-1$, let $r_{S,A} = F(K_A\{i-2\}, t-1)$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
       Else $\mathrm{VK}_A = \mathrm{VK}_{C,\mathsf{one}}$.
   (d) If $t \neq i-2$, let $r'_{S,A} = F(K_A\{i-2\}, t)$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (e) If $t \neq i-2$, $r'_{S,B} = F(K_B\{i-2\}, t)$. Compute $(\mathrm{SK}'_B, \mathrm{VK}'_B, \mathrm{VK}'_{B,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,B})$.
   (f) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
4. Computing next state and symbol (encrypted)
   (a) If $t = t^*$, output $b^*$.
   (b) If $i \leq t < t^*$, compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,3})$.
       Else do the following:
         i. Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
        ii. Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
       iii. Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
        iv. If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
         v. Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
5. Update accumulator, iterator and compute new signature
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = Reject$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$.
   (e) If $t = i-2$ and $m_{\mathrm{out}} = m_{i-2}$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}_C, m_{\mathrm{out}})$.
       Else if $t = i-2$ and $m_{\mathrm{out}} \neq m_{i-2}$ $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}_D, m_{\mathrm{out}})$.
       Else $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
6. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```

Figure 33: P_1

Hybrid H_2 In this hybrid, r_C and r_D are chosen uniformly at random; that is, the challenger computes $(\mathrm{SK}_C, \mathrm{VK}_C) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$ and $(\mathrm{SK}_D, \mathrm{VK}_D) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$.

Hybrid H_3 In this hybrid, the challenger computes constrained secret/verification keys. It computes $(\sigma_{C,\mathsf{one}}, \mathrm{VK}_{C,\mathsf{one}}, \mathrm{SK}_{C,\mathrm{abo}}, \mathrm{VK}_{C,\mathrm{abo}}) \leftarrow \mathsf{Spl.Split}(\mathrm{SK}_C, m_{i-2})$ and $(\sigma_{D,\mathsf{one}}, \mathrm{VK}_{D,\mathsf{one}}, \mathrm{SK}_{D,\mathrm{abo}}, \mathrm{VK}_{D,\mathrm{abo}}) \leftarrow \mathsf{Spl.Split}(\mathrm{SK}_D, m_{i-2})$. It then outputs an obfuscation of $P_3 = P_1\{i, t^*, K_E, K_A\{i-2\}, K_B\{i-2\}, \mathrm{VK}_{C,\mathsf{one}}, \sigma_{C,\mathsf{one}}, \mathrm{SK}_{D,\mathrm{abo}}, m_{i-2}\}$. Note that $\mathrm{SK}_C, \mathrm{VK}_C$, $\mathrm{SK}_D, \mathrm{VK}_D$ are not hardwired in this program.

Hybrid H_4 In this hybrid, the challenger chooses $\mathsf{P}_{\mathsf{Acc}}, w_0, \textsc{store}_0$ using $\mathsf{Acc.Setup\text{-}Enforce\text{-}Read}$. It then uses $\mathsf{P}_{\mathsf{Acc}}, w_0, \textsc{store}_0$, and proceeds as in previous experiment. It outputs an obfuscation of $P_1\{i, t^*, \mathsf{P}_{\mathsf{Acc}}, K_E, K_A\{i-2\}, K_B\{i-2\}, \mathrm{VK}_{C,\mathsf{one}}, \sigma_{C,\mathsf{one}}, \mathrm{SK}_{D,\mathrm{abo}}, m_{i-2}\}$.

Hybrid H_5 In this hybrid, the challenger makes SSB hash enforcing at $\mathsf{tmf}_{\mathsf{aux}}(i-2)$. It receives $M, (x_1, x_2)$ from $\mathcal{A}$, sets $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(i-2)$, chooses $H_{\mathsf{aux}} \leftarrow \mathsf{SSB.Gen}(1^\lambda, 1^{|x_2|}, \mathsf{pos}_{\mathsf{aux}})$. It computes $h_{\mathsf{aux}} = H_{\mathsf{aux}}(x_2)$ and sets $\mathsf{ek} = (H_{\mathsf{aux}}, h_{\mathsf{aux}})$. The remaining experiment is same as H_4.

Hybrid H_6 In this hybrid, the challenger first computes ciphertexts ct_1 and ct_2 as described in $\mathsf{Hyb}'_{2,i}$.

It then outputs an obfuscation of $P_6 = P_6\{i, t^*, \mathsf{P}_{\mathsf{Acc}}, K_E, K_A\{i-2\}, K_B\{i-2\}, \mathrm{VK}_{C,\mathsf{one}}, \sigma_{C,\mathsf{one}}, \mathrm{SK}_{D,\mathrm{abo}}, m_{i-2}, \mathsf{ct}_1, \mathsf{ct}_2\}$ (defined in Figure 34). This program differs from P_1 for inputs corresponding to $t = i-1$. Instead of decrypting, computing the next state and then encrypting, the program uses the hardwired ciphertexts.

Hybrid H_6 In this experiment, the challenger uses normal setup for Acc (that is, $\mathsf{Acc.Setup}$) instead of $\mathsf{Acc.Setup\text{-}Enforce\text{-}Read}$.

Hybrid H_7 In this experiment, the challenger outputs an obfuscation of W'_{int}.

Analysis Let $\mathsf{Adv}^x_{\mathcal{A}}$ denote the advantage of adversary $\mathcal{A}$ in hybrid H_x.

Claim D.19. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, for any PPT $\mathcal{A}$, $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^1_{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

Proof. In hybrid H_0, program P_0 is used, while in H_1, program P_1 is used. The only difference between the two programs is that P_1 uses punctured PRF keys $K_A\{i-2\}$ and $K_B\{i-2\}$. It also has the secret/verification keys computed using $F(K_A, i-2)$ and $F(K_B, i-2)$. As a result, using correctness of puncturable PRFs, it follows that the two programs have identical functionality. Therefore, by security of $i\mathcal{O}$, their obfuscations are computationally indistinguishable. ∎

Claim D.20. Assuming F is a selectively secure puncturable PRF, for any PPT $\mathcal{A}$, $|\mathsf{Adv}^1_{\mathcal{A}} - \mathsf{Adv}^2_{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

Proof. The proof of this claim follows from the selective security of puncturable PRF F. ∎

Claim D.21. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator and $\mathcal{S}$ satisfies $\mathrm{VK}_{\mathsf{one}}$ indistinguishability (Definition C.2), for any PPT $\mathcal{A}$, $|\mathsf{Adv}^2_{\mathcal{A}} - \mathsf{Adv}^3_{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

Proof. In order to prove this claim, we consider an intermediate hybrid program in which only the constrained secret keys $\sigma_{C,\mathsf{one}}$ and $\mathrm{SK}_{D,\mathrm{abo}}$ are hardwired, while VK_C is hardwired as the verification key. Using the security of $i\mathcal{O}$, we can argue that the intermediate step and H_2 are computationally indistinguishable. Next, we use $\mathrm{VK}_{\mathsf{one}}$ indistinguishability to show that the intermediate step and H_3 are computationally indistinguishable. ∎

Claim D.22. Assuming Acc satisfies indistinguishability of Read Setup (Definition C.5), for any PPT $\mathcal{A}$, $|\mathsf{Adv}^3_{\mathcal{A}} - \mathsf{Adv}^4_{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

Proof. The proof of this claim follows from Read Setup indistinguishability (Definition C.5). ∎

Claim D.23. Assuming SSB satisfies Definition B.1, for any PPT $\mathcal{A}$, $|\mathsf{Adv}^4_{\mathcal{A}} - \mathsf{Adv}^5_{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

P_6

```
Constants: Index $i$, Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_1, \mathsf{tmf}_2 \rangle$, time bound $T$, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E, K_A, K_B \in \mathcal{K}$, SSB Hash function $H_{\mathsf{aux}}$ and hash value $h_{\mathsf{aux}}$, $b^*, t^*$, $m_{i-2}$, $\mathrm{VK}_C$, $\sigma_C$, $\mathrm{SK}_D$, ciphertexts $\mathsf{ct}_1, \mathsf{ct}_2$.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value $v_{\mathsf{in}}$, signature $\sigma_{\mathrm{in}}$, accumulator proof $\pi$, SSB proof $\pi_{\mathsf{aux}}$, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. If $t > t^*$, output $\perp$.
3. Verifications
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) If $t \neq i-1$, let $r_{S,A} = F(K_A\{i-2\}, t-1)$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
       Else $\mathrm{VK}_A = \mathrm{VK}_{C,\mathsf{one}}$.
   (d) If $t \neq i-2$, let $r'_{S,A} = F(K_A\{i-2\}, t)$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (e) If $t \neq i-2$, $r'_{S,B} = F(K_B\{i-2\}, t)$. Compute $(\mathrm{SK}'_B, \mathrm{VK}'_B, \mathrm{VK}'_{B,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,B})$.
   (f) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
4. Computing next state and symbol (encrypted)
   (a) If $t = t^*$, output $b^*$.
   (b) If $i \leq t < t^*$, compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,3})$.
       Else if $t = i-1$ set $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{ct}_1$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{ct}_2$.
       Else do the following:
         i. Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
        ii. Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
       iii. Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
        iv. If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
         v. Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
5. Update accumulator, iterator and compute new signature
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$.
   (e) If $t = i-2$ and $m_{\mathrm{out}} = m_{i-2}$, $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}_C, m_{\mathrm{out}})$.
       Else if $t = i-2$ and $m_{\mathrm{out}} \neq m_{i-2}$ $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}_D, m_{\mathrm{out}})$.
       Else $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
6. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```

Figure 34: P_6

Proof. Note that the only difference between H_4 and H_5 is the enforcing index for SSB hash. In H_4, the hash is enforcing at $\mathsf{tmf}_{\mathsf{aux}}(i-3)$, while in H_5, it is enforcing at $\mathsf{tmf}_{\mathsf{aux}}(i-2)$. Using the index hiding property of SSB, we can argue that these two hybrids are computationally indistinguishable. ∎

Claim D.24. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, for any PPT $\mathcal{A}$, $|\mathsf{Adv}^5_{\mathcal{A}} - \mathsf{Adv}^6_{\mathcal{A}}| \leq \mathrm{negl}(\lambda)$.

Proof. We need to argue that the programs output in hybrids H_5 and H_6 are functionally identical. Let P_1 and P_2 denote these two programs. Note that the only difference is corresponding to input tuples with $t = i-1$. In P_1, for $t = i-1$, after the verification step, the ciphertexts $\mathsf{ct}_{\mathsf{sym},\text{in}}$ and $\mathsf{ct}_{\mathsf{st},\text{in}}$ are decrypted, the new state and symbol are computed, which are then encrypted. In H_6, the challenger computes the 'correct' state and symbol corresponding to $t = i-1$. It encrypts them (deterministic encryption using a PRF key), and hardwires the ciphertexts. If $t = i-1$ and the verifications pass, then the program P_2 does not decrypt input ciphertexts; instead it sets $\mathsf{ct}_{\mathsf{sym},\text{out}}$ and $\mathsf{ct}_{\mathsf{st},\text{out}}$ to be the hardwired ciphertexts $\mathsf{ct}_1, \mathsf{ct}_2$.

At time step $t = i-2$, both programs output 'A' type signatures for the correct message m_{i-2}. Threfore, at time $t = i-2$, there is exactly one input tuple that is accepted, which is m_{j-2}, and hence $m_{\text{in}} = m_{i-2}$, which implies $\mathsf{ct}_{\mathsf{sym},\text{in}}$ is the correct state's encryption. Since the SSB hash and accumulator are appropriately enforced, $\mathsf{sym}_{\mathsf{aux}}$ and $\mathsf{ct}_{\mathsf{sym},\text{in}}$ are the correct symbols input to P_1. As a result, the new state/symbol are also the correct state/symbol respectively, and hence P_1 outputs $\mathsf{ct}_1, \mathsf{ct}_2$. Therefore, both programs have identical behavior on all inputs. ∎

Claim D.25. Assuming Acc satisfies indistinguishability of Read Setup (Definition C.5), for any PPT $\mathcal{A}$, $|\mathsf{Adv}^5_{\mathcal{A}} - \mathsf{Adv}^6_{\mathcal{A}}| \leq \text{negl}(\lambda)$.

Proof. This step is reverse of the step from H_3 to H_4. ∎

Claim D.26. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF and $\mathcal{S}$ satisfies VK_{one} indistinguishability (Definition C.2), for any PPT $\mathcal{A}$, $|\mathsf{Adv}^7_{\mathcal{A}} - \mathsf{Adv}^8_{\mathcal{A}}| \leq \text{negl}(\lambda)$.

This step is the reverse of the step from H_0 to H_3. Therefore, using similar intermediate hybrid experiments, a similar proof works here as well. ∎

Lemma D.11. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, F is a selectively secure puncturable PRF, Itr is an iterator satisfying Definitions C.9 and C.10, Acc is an accumulator satisfying Definitions C.5, C.6, C.7 and C.8, $\mathcal{S}$ is a splittable signature scheme satisfying security Definitions C.1, C.2, C.3 and C.4, $|\mathsf{Adv}'^{int}_{\mathcal{A}} - \mathsf{Adv}'^{2,i}_{\mathcal{A}}| \leq \text{negl}(\lambda)$.

The proof of this lemma is similar to the proof of Lemma D.9.

Combining Lemma D.9, Lemma D.10 and Lemma D.11, we get our desired proof for Lemma D.2.

D.3 Proof of Lemma D.3

We will first define hybrids $H_0, \ldots, H_5$, where H_0 corresponds to $\mathsf{Hyb}'_{2,i}$ and H_5 corresponds to $\mathsf{Hyb}_{2,i-1}$.

Hybrid H_0 This corresponds to $\mathsf{Hyb}'_{2,i}$.

Hybrid H_1 In this hybrid, the challenger punctures the PRF key K_E on inputs corresponding to $t = i-1$. It outputs an obfuscation of program $W_1 = \mathsf{Prog}'\text{-}2\text{-}i\text{-}1\{i, t^*, K_E\{i-1\}, K_A, \mathsf{ct}_1, \mathsf{ct}_2\}$ where $\mathsf{Prog}'\text{-}2\text{-}i\text{-}1$ is defined in Figure 35. Note that the only difference between $\mathsf{Prog}'\text{-}2\text{-}i$ and $\mathsf{Prog}'\text{-}2\text{-}i\text{-}1$ is that the latter uses a punctured PRF key $K_E\{i-1\}$ instead of K_E.

Hybrid H_2 In this hybrid, the challenger computes $\mathsf{sk} \leftarrow \mathsf{Setup}_{\text{PKE}}(1^\lambda)$ using true randomness. Also, the ciphertexts ct_1 and ct_2 are computed using true randomness; that is, $\mathsf{ct}_1 \leftarrow \mathsf{SKE.Enc}(\mathsf{sk}, \mathsf{sym}^*)$ and $\mathsf{ct}_2 \leftarrow \mathsf{SKE.Enc}(\mathsf{sk}, \mathsf{st}^*)$.

Hybrid H_3 In this hybrid, the challenger sets $\mathsf{ct}_1 = \mathsf{SKE.Enc}(\mathsf{sk}, \mathsf{erase})$ and $\mathsf{ct}_2 = \mathsf{SKE.Enc}(\mathsf{sk}, \mathsf{erase})$.

```
Prog′-2-$i$-1

Constants: Index $i$, Turing machine $M = \langle Q, \Sigma_{\mathsf{tape}}, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}, \mathsf{tmf}_1, \mathsf{tmf}_2 \rangle$, time bound $T$, Public parameters for accumulator $\mathsf{P}_{\mathsf{Acc}}$, Public parameters for Iterator $\mathsf{P}_{\mathsf{Itr}}$, Puncturable PRF keys $K_E\{i-1\}, K_A \in \mathcal{K}$, SSB Hash function $H_{\mathsf{aux}}$ and hash value $h_{\mathsf{aux}}$, $b^*, t^*$, $\mathsf{ct}_1$, $\mathsf{ct}_2$.

Input: Time $t \in [T]$, encrypted symbol and last-write time $(\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w})$, auxiliary tape symbol $\mathsf{sym}_{\mathsf{aux}}$, encrypted state $\mathsf{ct}_{\mathsf{st},\mathrm{in}}$, accumulator value $w_{\mathrm{in}} \in \{0,1\}^{\ell_{\mathsf{Acc}}}$, Iterator value $v_{\mathsf{in}}$, signature $\sigma_{\mathrm{in}}$, accumulator proof $\pi$, SSB proof $\pi_{\mathsf{aux}}$, auxiliary value $\mathsf{sym}_{\mathsf{aux}}$.

1. Let $\mathsf{pos}_{\mathrm{in}} = \mathsf{tmf}_{\mathrm{wk}}(t-1)$, $\mathsf{pos}_{\mathsf{aux}} = \mathsf{tmf}_{\mathsf{aux}}(t-1)$ and $\mathsf{pos}_{\mathrm{out}} = \mathsf{tmf}_{\mathrm{wk}}(t)$.
2. If $t > t^*$, output $\perp$.
3. Verifications
   (a) If $\mathsf{Acc.Verify\text{-}Read}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{in}}, \ell\text{-}\mathsf{w}), \mathsf{pos}_{\mathrm{in}}, \pi) = 0$ or $\ell\text{-}\mathsf{w} \geq t$, output $\perp$.
   (b) If $\mathsf{SSB.Verify}(H_{\mathsf{aux}}, h_{\mathsf{aux}}, \mathsf{pos}_{\mathsf{aux}}, \mathsf{sym}_{\mathsf{aux}}, \pi_{\mathsf{aux}}) = 0$, output $\perp$.
   (c) Let $F(K_A, t-1) = r_{S,A}$. Compute $(\mathrm{SK}_A, \mathrm{VK}_A, \mathrm{VK}_{A,\mathsf{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{S,A})$.
   (d) Let $m_{\mathrm{in}} = (v_{\mathsf{in}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}})$. If $\mathsf{Spl.Verify}(\mathrm{VK}_A, m_{\mathrm{in}}, \sigma_{\mathrm{in}}) = 0$ output $\perp$.
4. Computing next state and symbol (encrypted)
   (a) If $t = t^*$, output $b^*$.
   (b) If $i \leq t < t^*$ compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E\{i-1\}, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{erase}; r_{t,3})$.
       Else if $t = i-1$ set $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{ct}_1$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{ct}_2$.
       Else do the following:
         i. Let $(r_{\ell\text{-}\mathsf{w},1}, r_{\ell\text{-}\mathsf{w},2}, r_{\ell\text{-}\mathsf{w},3}) = F(K_E\{i-1\}, \ell\text{-}\mathsf{w})$, $\mathsf{sk}_{\ell\text{-}\mathsf{w}} = \mathsf{SKE.Setup}(1^\lambda; r_{\ell\text{-}\mathsf{w},1})$, $\mathsf{sym} = \mathsf{SKE.Dec}(\mathsf{sk}_{\ell\text{-}\mathsf{w}}, \mathsf{ct}_{\mathsf{sym},\mathrm{in}})$.
        ii. Let $(r_{t-1,1}, r_{t-1,2}, r_{t-1,3}) = F(K_E\{i-1\}, t-1)$, $\mathsf{sk}_{\mathsf{st}} = \mathsf{SKE.Setup}(1^\lambda, r_{t-1,1})$, $\mathsf{st} = \mathsf{SKE.Dec}(\mathsf{sk}_{\mathsf{st}}, \mathsf{ct}_{\mathsf{st},\mathrm{in}})$.
       iii. Let $(\mathsf{st}', \mathsf{sym}', \beta) = \delta(\mathsf{st}, \mathsf{sym}, \mathsf{sym}_{\mathsf{aux}})$.
        iv. If $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{rej}}$ output 0. Else if $\mathsf{st}_{\mathrm{out}} = q_{\mathrm{acc}}$ output 1.
         v. Compute $(r_{t,1}, r_{t,2}, r_{t,3}) = F(K_E\{i-1\}, t)$, $\mathsf{sk}' = \mathsf{SKE.Setup}(1^\lambda; r'_{t,1})$, $\mathsf{ct}_{\mathsf{sym},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{sym}'; r_{t,2})$ and $\mathsf{ct}_{\mathsf{st},\mathrm{out}} = \mathsf{SKE.Enc}(\mathsf{sk}', \mathsf{st}'; r_{t,3})$.
5. Update accumulator, iterator and compute new signature
   (a) Compute $w_{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{P}_{\mathsf{Acc}}, w_{\mathrm{in}}, (\mathsf{ct}_{\mathsf{sym},\mathrm{out}}, t), \mathsf{pos}_{\mathrm{in}}, \mathsf{acc\text{-}aux})$. If $w_{\mathrm{out}} = \mathit{Reject}$, output $\perp$.
   (b) Compute $v_{\mathsf{out}} = \mathsf{Itr.Iterate}(\mathsf{P}_{\mathsf{Itr}}, v_{\mathsf{in}}, (\mathsf{ct}_{\mathsf{st},\mathrm{in}}, w_{\mathrm{in}}, \mathsf{pos}_{\mathrm{in}}))$.
   (c) Let $F(K_A, t) = r'_{S,A}$. Compute $(\mathrm{SK}'_A, \mathrm{VK}'_A, \mathrm{VK}'_{A,\mathsf{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r'_{S,A})$.
   (d) Let $m_{\mathrm{out}} = (v_{\mathsf{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, \mathsf{pos}_{\mathrm{out}})$ and $\sigma_{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{SK}'_A, m_{\mathrm{out}})$.
6. Output $\mathsf{pos}_{\mathrm{in}}, \mathsf{ct}_{\mathsf{sym},\mathrm{out}}, \mathsf{ct}_{\mathsf{st},\mathrm{out}}, w_{\mathrm{out}}, v_{\mathsf{out}}, \sigma_{\mathrm{out}}$.
```
Figure 35: Prog′-2-i-1

Hybrid H_4 In this hybrid, the challenger computes the ciphertexts using pseudorandom strings generated using $F(K_E, \cdot)$. More precisely, the challenger computes $(r_{i-1,1}, r_{i-1,2}, r_{i-1,3}) = F(K_E, i-1)$, $\mathsf{sk} \mathsf{SKE.Setup}(1^\lambda; r_{i-1,1})$, $\mathsf{ct}_1 = \mathsf{SKE.Enc}(\mathsf{sk}, \mathsf{erase}; r_{i-1,2})$ and $\mathsf{ct}_2 = \mathsf{SKE.Enc}(\mathsf{sk}, \mathsf{erase}; r_{i-1,3})$.

Hybrid H_5 This corresponds to $\mathsf{Hyb}_{2,i-1}$.

D.3.1 Analysis

Claim D.27. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, for any PPT adversary $\mathcal{A}$, $\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^1_{\mathcal{A}} \leq \mathrm{negl}(\lambda)$.

Proof. To prove this claim, it suffices to show that W_0 and W_1 are functionally identical. The crucial observation for this proof is the fact that $F(K_E, i-1)$ is not used anywhere in program W_0. For inputs corresponding to $t > i-1$, both programs don't use $F(K_E, i-1)$ since the programs do not decrypt for $t > i-1$. For $t = i-1$, the ciphertexts ct_1 and ct_2 are hardwired. For $t < i-1$, note that it only computes $F(K_E, \tau)$ for $\tau < i-1$. As a result, $F(K_E, \cdot)$ is not evaluated at input $i-1$, and therefore, it is safe to puncture K_E on input $i-1$ without affecting functionality. The rest follows from the correctness of puncturable PRFs. ∎

Claim D.28. Assuming F is a selectively secure puncturable PRF, for any PPT adversary $\mathcal{A}$, $\mathsf{Adv}^1_{\mathcal{A}} - \mathsf{Adv}^2_{\mathcal{A}} \leq \text{negl}(\lambda)$.

Proof. The proof of this claim is similar to the proof of Claim **??**; it follows from the selective security of puncturable PRF F. ∎

Claim D.29. Assuming PKE is $\mathsf{IND\text{-}CPA}$ secure, for any PPT adversary $\mathcal{A}$, $\mathsf{Adv}^2_{\mathcal{A}} - \mathsf{Adv}^3_{\mathcal{A}} \leq \text{negl}(\lambda)$.

Proof. Note that the secret key sk is not required in both hybrids H_2 and H_3. Suppose there exists an adversary $\mathcal{A}$ that can distinguish between H_2 and H_3 with advantage ϵ. Then we can construct a PPT algorithm $\mathcal{B}$ that breaks the $\mathsf{IND\text{-}CPA}$ security of SKE with advantage ϵ. The reduction algorithm interacts with $\mathcal{A}$ and computes $\mathsf{sym}^*, \mathsf{st}^*$. It sends $\boldsymbol{m_0} = (\mathsf{sym}^*, \mathsf{st}^*)$ and $\boldsymbol{m_1} = (\mathsf{erase}, \mathsf{erase})$ as the challenge message pairs, and receives a ciphertext pair $(\mathsf{ct}_1, \mathsf{ct}_2)$. $\mathcal{B}$ can now perfectly simulate H_2 or H_3 for $\mathcal{A}$, depending on whether $(\mathsf{ct}_1, \mathsf{ct}_2)$ are encryptions of $\boldsymbol{m_0}$ or $\boldsymbol{m_1}$. This completes our proof. ∎

Claim D.30. Assuming F is a selectively secure puncturable PRF, for any PPT adversary $\mathcal{A}$, $\mathsf{Adv}^3_{\mathcal{A}} - \mathsf{Adv}^4_{\mathcal{A}} \leq \text{negl}(\lambda)$.

Proof. This step is the reverse of the step from H_1 to H_2; its proof also follows from selective security of puncturable PRFs. ∎

Claim D.31. Assuming $i\mathcal{O}$ is a secure indistinguishability obfuscator, for any PPT adversary $\mathcal{A}$, $\mathsf{Adv}^4_{\mathcal{A}} - \mathsf{Adv}^5_{\mathcal{A}} \leq \text{negl}(\lambda)$.

Proof. The only difference between the programs used in the two hybrids is that one uses a punctured key $K_E\{i-1\}$, while the other uses K_E. Using the correctness of puncturable PRFs, we can argue that they are functionally identical. As a result, from the security of $i\mathcal{O}$, their obfuscations are computationally indistinguishable. ∎

D.4 Proof of Lemma D.4

Proof. The only difference between hybrid $\mathsf{Hyb}_{2,1}$ and Hyb_3 is with respect to the encoding of the input. In the former case, the challenger computes encryption of input x_1 (followed by hashing using accumulator, signing etc.), while in the latter case, the challenger computes encryption of $\mathsf{erase}^{|x_1|}$. To prove this lemma, we will introduce two hybrid experiment H_1 and H_2.

The experiment H_1 is similar to $\mathsf{Hyb}_{2,1}$, except that the obfuscated program uses a punctured PRF key $K_E\{0\}$ for outputting encryptions, and it has encryption of starting state and starting symbol hardwired. Since this program does not compute $F(K_E, 0)$, these two programs are functionally identical, and hence the hybrid experiments $\mathsf{Hyb}_{2,1}$ and H are computationally indistinguishable.

In experiment H_2, the challenger computes uses a truly random string instead of $F(K_E, 0)$. Using the PRF security, we can argue that H_1 and H_2 are computationally indistinguishable.

Finally, using the security of the encryption scheme, we can show that H_2 and Hyb_3 are computationally indistinguishable. This concludes our proof.

∎