From Signature-Based Witness Encryption to RAM Obfuscation: Achieving Blockchain-Secured Cryptographic Primitives

Lev Stambler
levstamb@umd.edu

Department of Computer Science, University of Maryland
Neon Tetra LLC

June 06, 2025

Abstract

Goyal and Goyal demonstrated that extractable witness encryption, when combined with smart-contract equipped proof-of-stake blockchains, can yield powerful cryptographic primitives such as one-time programs and pay-to-use programs. However, no standard model construction for extractable witness encryption is known, and instantiations from alternatives like indistinguishability obfuscation are highly inefficient.

This paper circumvents the need for extractable witness encryption by combining signature-based witness encryption (Döttling et al.) with witness encryption for KZG commitments (Fleischhacker et al.). Inspired by Goyal et al., we introduce $T+1$-Extractable Witness Encryption for Blockchains ($T+1$-eWEB), a novel primitive that encrypts a secret, making its decryption contingent upon the subsequent block's state. Leveraging $T+1$-eWEBs, we then build a conditional one-time memory, leading to a $T+1$ one-time program ($T+1$-OTP) also conditional on the next block state. Finally, using our $T+1$-OTP, we develop a conditional RAM obfuscation scheme where program execution can be contingent on the blockchain state, thereby enabling applications like pay-to-use programs.

Despite its theoretical value, our construction is impractical due to a "bit-by-bit" signing requirement for the state root and an inefficient method for storing validator keys. We thus posit the construction of a practical $T+1$-OTP as a significant open problem. This work provides the first theoretical pathway for building such primitives without extractable witness encryption, representing a novel step for blockchain-secured cryptography

1 Introduction

The celebrated Bitcoin paper provided a novel approach to solving a long standing set of questions in distributed systems, opening the floodgates to cryptocurrencies [1]. The key

innovation of Bitcoin is an economic mechanism, rather than a purely computational one, that allows a set of nodes to agree on a single state of the system.

A line of works, started by Goyal and Goyal, asks a similar question for cryptographic primitives which are either impossible (such as obfuscation and one-time programs) or are challenging to achieve in a practical way [2]. Specifically, the work asks whether we can use blockchains to achieve cryptographic primitives which are *as secure* as the blockchain itself?

Though various primitives have been discussed in the literature, we will highlight one of them, **RAM obfuscation**, which is essentially the most powerful primitive we can hope to achieve in a cryptographic setting: obfuscation of a program with its own internally memory [3].

Ref. [2] uses a very advanced primitive, extractable witness encryption, to build a blockchain based one-time program and then a variant of RAM obfuscation. However, very little is known about extractable witness encryption[1]. Thus, a line of works "emulate" extractable witness encryption using a distributed system armed with proactive secret sharing [4–8].

Unfortunately though, these new systems do not easily integrate into existing blockchains, and thus, to be considered secure (from an crypto-economic perspective), need to bootstrap a new blockchain or convince an existing blockchain to adopt a highly complex system[2].

We are then left with a conundrum: how can we build primitives such as one-time programs and RAM obfuscation that are as secure as the blockchain itself but also have the flexibility to be used in a wide variety of applications?

1.1 Our Contributions

Rather than trying to build a generic system for all proof of stake blockchains, we will focus on a blockchain which does three things:

1. Commit to its state via a KZG commitment [9].
2. Have the signing and verification of its committed state be done via a modified BLS signature [10].
3. Include its set of validators for the next block in the current block.

We then show how, via minor modifications to the BLS signature scheme, we can build a new primitive which we term **T + 1-eWEB** (T + 1-extractable witness encryption for blockchains). eWEBs were introduced by Goyel et al. [4] and allow for "encrypting" a secret to the blockchain and decrypting the secret if and only if a certain condition, relative to the blockchain state, is met. Our primitive is similar but only allows for decryption to occur if the condition is met at the block subsequent to the block in which the secret was encrypted. Importantly, our $T+1$-eWEBs are *as secure as the blockchain itself.* In order to decrypt the secret without the condition being met, an adversary must be able (to at least partially) forge (a signature on) the blockchain state! We present an illustration of $T+1$-eWEBs in Figure 1.

[1] To the author's knowledge, it is still not known if extractable witness encryption is possible in the standard model.

[2] Most blockchains with a large market cap have a level of inherent "technical conservatism" as technical overhauls are quite difficult to achieve, have unintended consequences, and risk the entire system. It is therefore not surprising that leading blockchains have not adopted these systems.

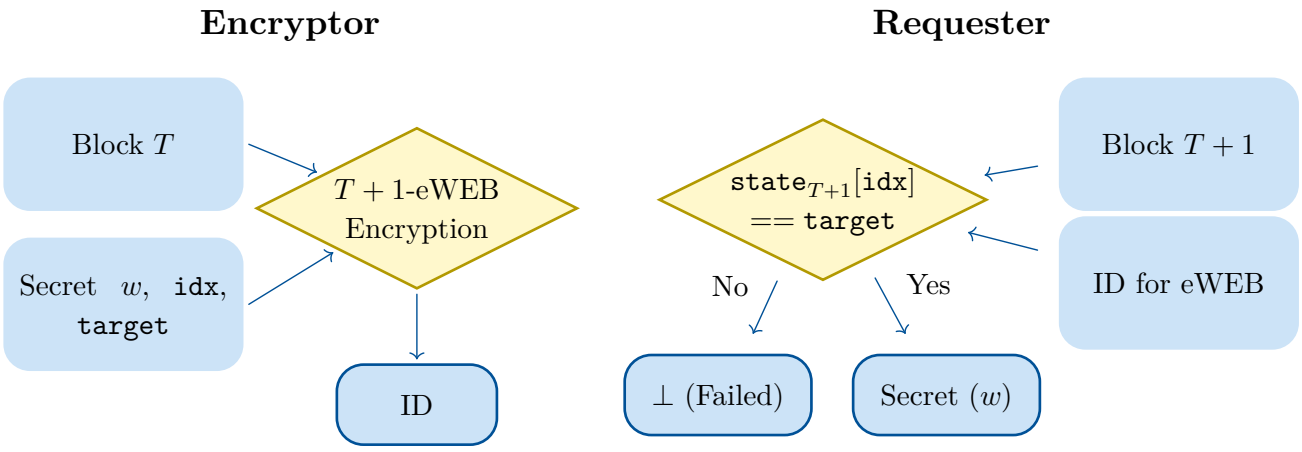

Figure 1: An illustration of the $T+1$-eWEB mechanism. The left panel shows the setup phase, where the mechanism outputs an identifier that will be used for retrieval. The right panel shows the reveal phase, where the requester checks the condition on the blockchain state and retrieves the secret if the condition is met.

Though $T+1$-eWEBs may seem limited, we show that they are quite powerful and can be used to build a variety of primitives, including (but not limited to):

- $T+1$ Conditional one-time programs, which allow a program to be executed exactly once, contingent on a future blockchain state satisfying a condition. We note that this is a more general version of a $T+1$ pay-to-use program, which was introduced in Ref. [2].
- RAM obfuscation: an obfuscation of a program P which has its own (encrypted) internal memory *which cannot be rewound*. Moreover, the execution of P can be contingent on some condition on the blockchain state as well.

Interestingly, our construction of RAM obfuscation only requires interacting with the blockchain when a user wants to execute the program and does not suffer from a "liveness" assumption which is stronger than the underlying blockchain's liveness assumptions. Unfortunately though, our construction requires the blockchain to store its set of validator signatures in a specific, high-overhead way, which is not the case for most blockchains today[3].

1.2 Technical Overview

Our construction is based on the following primitives: signature-based witness encryption (SWE), introduced in the McFly work [11] and extractable witness encryption for KZG commitments (WE-KZG) [12]. We will first briefly review these primitives and then outline our constructions.

SWE and WE-KZG

At a high level, SWE allows for encrypting a message m towards a signing committee V, threshold t, and tag T such that the encrypted ciphertext can only be decrypted given an aggregate signature of parties $U \subseteq V$ on tag T with $|U| \geq t$. In other words, a message can only be decrypted given a valid aggregate signature to a pre-specified tag. The construction presented in the McFly protocol [11] constructs SWE based on BLS-type signatures.

[3]The verification keys for each validator must be stored in a "bit-by-bit" way, where each individual bit of the verification key is stored in a specific data-slot on the blockchain. We leave the removal of this requirement to future work.

WE-KZG on the other hand, allows for encrypting a secret s under a KZG commitment $\texttt{comm}$ to state, $\texttt{state}$, such that s can only be recovered via a valid opening π to $\texttt{state}[\texttt{idx}] = \texttt{target}$ for $\texttt{idx}, \texttt{target}$ set by the encryptor. In other words, WE-KZG allows for encrypting a message to a specific opening to a commited vector.

High Level Idea and $T+1$-eWEBs

Key to this work, we change the BLS signature scheme outlined in the McFly protocol to sign a KZG commitment "bit-by-bit" rather than as a whole. So, if we have a commitment $\texttt{comm}$ to the blockchain state $\texttt{state}_T$, the signature does not sign $H(T \parallel \texttt{comm})$ for hash function H but rather signs $H(T \parallel i \parallel \texttt{comm}_i)$ for each bit i of the commitment. This modification allows us to use signature-based witness encryption alongside the fact that only *one* valid signature is produced per block to create a sort of "one-time" program for the *blockchain commitment*. Specifically, using garbled circuits, we can garble a circuit $C(\texttt{comm})$ which takes as input the KZG commitment $\texttt{comm}$. The garbling process then outputs wire labels $w_1^0, w_1^1, ..., w_{|\texttt{comm}|}^0, w_{|\texttt{comm}|}^1$. Then, we can encrypt the wire labels such that a user can only decrypt w_i^b if a valid blockchain signature, σ_i, is produced for message $H(T+1 \parallel i \parallel \texttt{comm}_i)$. Note that if the blockchain is secure, we never should have a signature be produced for both $H(T+1 \parallel i \parallel 0)$ and $H(T+1 \parallel i \parallel 1)$. Thus, after the blockchain signs the state commitment $\texttt{comm}$ for the $T+1$-th state, the garbled circuit can be executed on input $\texttt{comm}$ but should never be able to execute on $\texttt{comm}' \neq \texttt{comm}$.

At first glance, using SWE to encrypt a one-time program may seem to have a narrow use-case, but we now show how to use our second key primitive, extractable witness encryption for KZG commitments, which will allow us to make much more powerful primitives. Recall that extractable witness encryption for KZG commitments allows for encrypting a message towards a commitment to a state v, opening index i, and target t to get a ciphertext $\texttt{ct}_{\text{KZG}}$. Then, $\texttt{ct}_{\text{KZG}}$ can only be decrypted given a valid opening, π, of the commited state proving that $v[i] = t$. Armed with this primitive, we can now build build our $T+1$-eWEB for the following simple condition: for the next state, $\texttt{state}_{T+1}$ of a blockchain, we decrypt if $\texttt{state}_{T+1}[\texttt{idx}] = \texttt{target}$ for some predefined index $\texttt{idx}$ and target value $\texttt{target}$. Note that we cannot simply provide $\texttt{ct}_{\text{KZG}}$ for the $T+1$-th state commitment of the blockchain as, at time T, we *do not know* what the $T+1$-th state will be! Instead, we use the previously outlined one-time program technique where the one-time program will "create" the ciphertext $\texttt{ct}_{\text{KZG}}$ given the state commitment $\texttt{comm}_{T+1}$. We provide a high-level illustration of our the setup in our construction in Figure 2 and the secret release in Figure 3.

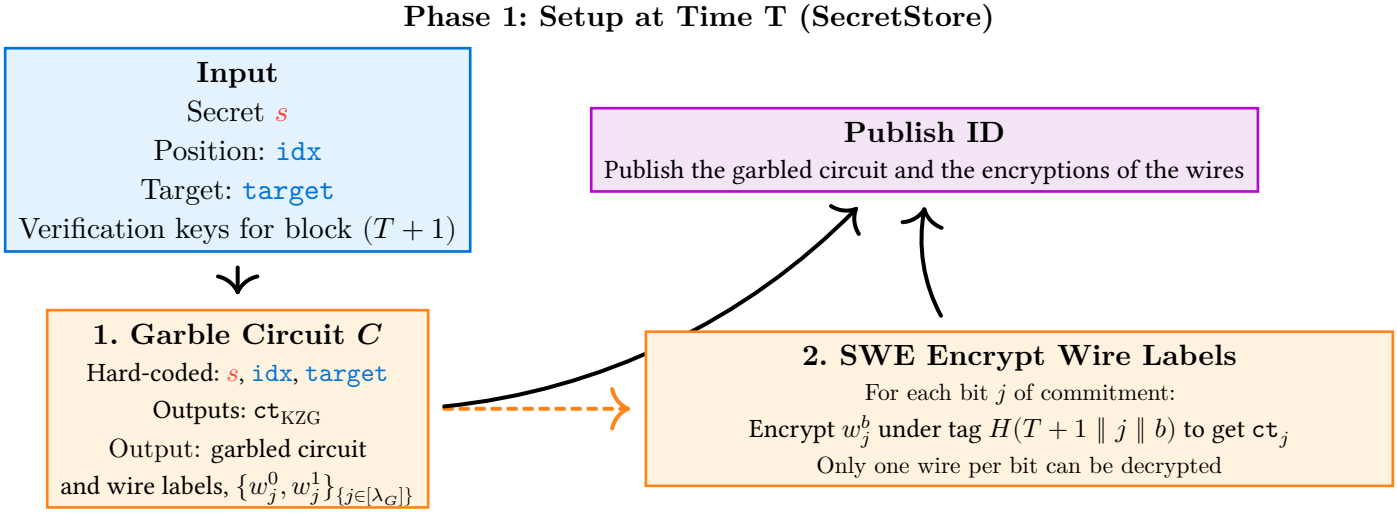

Figure 2: A visualization of the setup phase for our $T+1$-eWEB scheme.

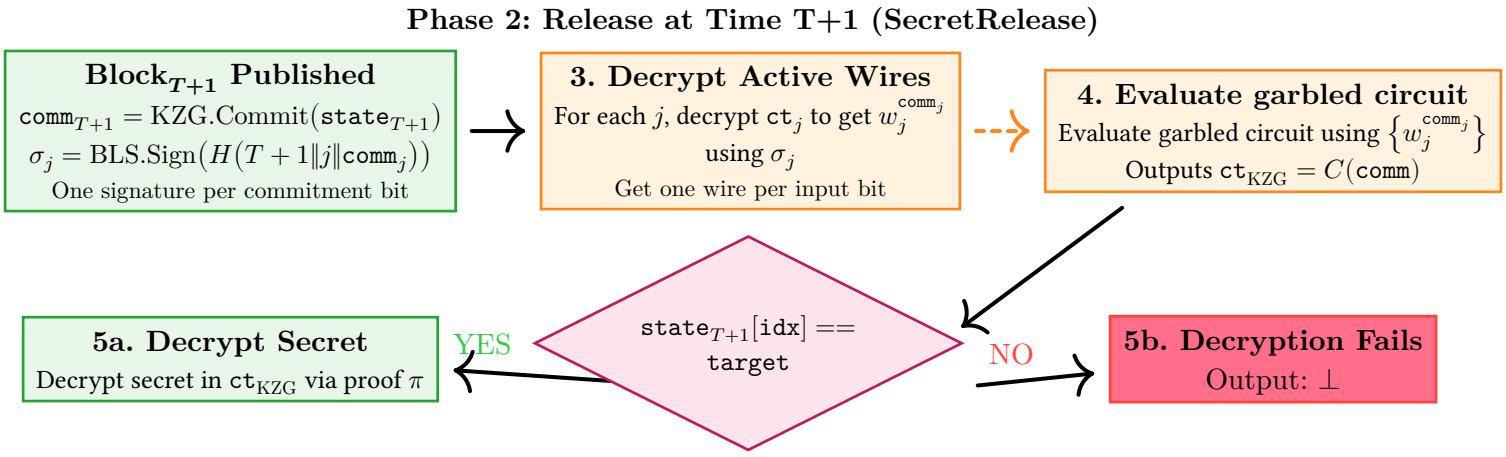

Figure 3: A visualization of the release phase for our $T+1$-eWEB scheme.

Constructing One-Time Programs from $T+1$-eWEBs

Recall that a $T+1$-eWEB allows us to store a secret such that the secret can only be retrieved if a specific data-slot on the blockchain has a target value. Given that a data-slot can only have *one* value for the $T+1$-th block[4], we can use our $T+1$-eWEBs two store two messages, where at most one of the messages can be decoded. In more detail, we would use our eWEB to encrypt m_0 requiring $\mathtt{state}_{T+1}[\mathtt{idx}] = 0$ to decrypt and encrypt m_1 requiring $\mathtt{state}_{T+1}[\mathtt{idx}] = 1$ to decrypt.

We now have built a *one-time memory* where only one-message can be decrypted. As shown by Goldwasser et al. [13], we can combine one-time memories with garbled circuits to then get one-time programs. And so, we can get $T+1$-one time programs, where we can evaluate the program exactly once and only if the program's input is contained within the $T+1$-th state of the blockchain.

Though an interesting primitive in its own right, $T+1$-OTPs are still not as powerful as we would like. Particularly, we require that the program's inputs are posted to the blockchain

[4]Otherwise, we would have *two different* (valid) blockchain states at block $T+1$, breaking the blockchain's guarantee of a unique and correct state.

before the $T+1$-th block which is a sort of strong liveness assumption and decreases the OTP's usefulness.

We thus modify our construction to allow for the program's input to be posted *after* the $T+1$ -th block. The key idea is to use our $T+1$-OTPs in a "recursive way." At a high level, say we are encrypting circuit $C : \{0,1\}^r \to \mathcal{Y}$. We will first modify C to a circuit $C' : \{\bot\} \cup \{0,1\}^r \to \mathcal{Y}$. We then set $\texttt{state}_T[\texttt{idx}_i] = \bot$ for $i \in [r]$. Then, if the input is set to $\bot$, we will have $C'(\bot)$ output a one-time program for block $T+2$. If the input is set to $x \in \{0,1\}^r$, we will have $C'(x)$ output $C(x)$. In other words, if the input is not set prior to the $T+1$-th block, the program will "refresh" itself to allow for new inputs to be set prior to the $T+2$-th block. And, given that $\texttt{state}_T[\texttt{idx}_i] = \bot$ by default, if the state is not set prior to the $T+2$-th block, the $T+2$ -OTP will again refresh itself to allow for new inputs to be set prior to the $T+3$-th block and so on. In this way, we can have a program which can be executed at any time, but only once after block $T+j$, and only if the program's input is set prior to the finalization of the $T+j$-th block[5][6].

We provide a high-level illustration of our construction in Figure 4.

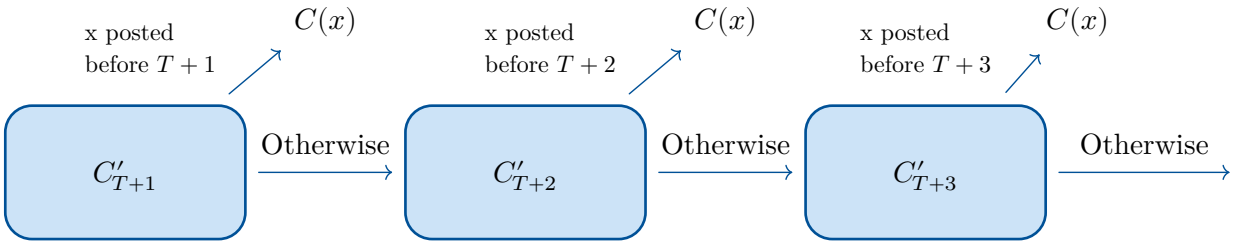

Figure 4: A visualization of our one-time program construction. If input x is set before block $T+1$, then we can evaluate to $C(x)$. If not, the program will "refresh" itself to output a one-time program for C'_{T+2}.

Constructing RAM Obfuscation

Now that we have our fully fledged one-time program, it is pretty straight forward to build RAM obfuscation. We can think of RAM obfuscation as an obfuscation of a circuit $C : \mathcal{X} \times \mathcal{M} \to \mathcal{Y} \times \mathcal{M}$ where $\mathcal{M}$ represents the space of memory. Then, an obfuscation of C for initial memory RAM_0, allows for evaluating $C(\cdot, \text{RAM}_0)$ *once* to get next memory RAM_1 and output y. Then, the obfuscation allows for evaluating $C(\cdot, \text{RAM}_1)$ once and so on. So, we start by publishing a one-time program for $\overline{C}(\cdot, \text{RAM}_0)$ where $\overline{C}(x, \text{RAM}_0)$ will output y for $y, \text{RAM}_1 = C(x, \text{RAM}_0)$ as well as the one-time program for $\overline{C}(\cdot, \text{RAM}_1)$. We can then see that this process continues allowing for the program to be executed once and only once for each memory state while still allowing for the "next" execution. We provide a high-level illustration of our construction in Figure 5.

[5]Our above exposition is slightly simplified. Within the main body of our work, we actually separate out whether the program's input is set or not into a separate bit so that now $C' : \{0,1\}^r \times \{0,1\} \to \mathcal{Y}$. This final bit acts as our flag bit and decrypts to 0 if the smart contract's last set block is less than the current block index. If the block is set at time T, then the current block index on the smart contract gets set to T and so we can decrypt to the flag 1 if the program's input is set.

[6]Within our construction, we do not separately construct an OTP scheme and RAM obfuscation but rather instantiate RAM obfuscation directly from our $T+1$-OTPs; the construction of general OTPs then follows from the construction of RAM obfuscation.

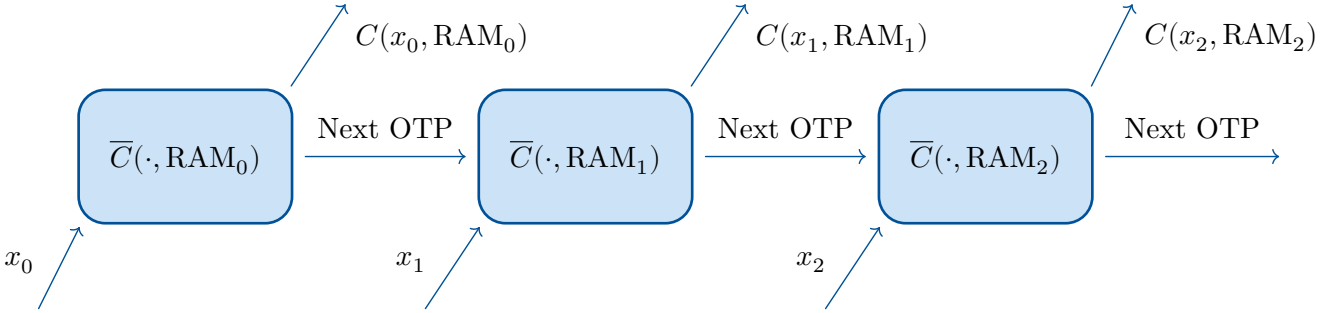

Figure 5: A visualization of our RAM obfuscation construction from our one-time programs. $\overline{C}(\cdot, \mathrm{RAM}_i)$ allows for evaluating $C(x, \mathrm{RAM}_i)$ once and only once and also outputs the next one-time program for $\overline{C}(\cdot, \mathrm{RAM}_{i+1})$.

Smart Contract Gating, Pay to Evaluate, and Variations on our Results
As noted by Goyal and Goyal [2], our use of the blockchain alongside obfuscation allows for a variety of new applications. Specifically, because program evaluation (for our OTPs and RAM obfuscation) is contingent on a blockchain state, we can have the setting of the program input be controlled by a *smart contract.* And so, we can require one of many possible conditions to be met before the program can be executed. For example, we can require that the program's input is set by a smart contract which requires a payment to be made as pointed out by Ref. [2]. This allows for a new class of "pay to evaluate" programs. We can also require that a certain number of (encrypted) transactions are posted to the blockchain prior to the program being executed, allowing for dark pools and other MEV-free financial applications.

We only skim the surface of the applications of our primitive, but we note that the primitives we build are extremely flexible and powerful.

1.3 Outline

In Section 2, we provide brief overviews of the cryptographic primitives on which we rely. In Section 3, we give an overview of our "blockchain" formalization, though not the same exact specification as the Ethereum blockchain, we aim to be as close as possible. Then, we introduce our $T+1$-eWEB construction in Section 4 and use the construction to build one-time programs in Section 5. Using the one-time program construction, we then build RAM obfuscation in Section 6. Finally, we conclude our work in Section 7.

2 Preliminaries

We now proceed to define the primitives which we will use in this work.

2.1 KZG Polynomial Commitments and BLS Signatures

Definition 2.1 *(KZG Polynomial Commitments [9])*: A KZG polynomial commitment scheme over field $\mathbb{F}_p$ consists of four algorithms $(\mathrm{Setup}, \mathrm{Commit}, \mathrm{Open}, \mathrm{Verify})$:

- $\mathtt{ck} \leftarrow \mathrm{Setup}(1^\lambda, 1^d)$: Generates a commitment key for polynomials of degree at most d.
- $\mathtt{comm} \leftarrow \mathrm{Commit}(\mathtt{ck}, f)$: Commits to a polynomial $f \in \mathbb{F}_p[X]$.
- $\pi \leftarrow \mathrm{Open}(\mathtt{ck}, f, \alpha, \beta)$: Generates an opening proving $f(\alpha) = \beta$.
- $b \leftarrow \mathrm{Verify}(\mathtt{ck}, \mathtt{comm}, \pi, \alpha, \beta)$: Verifies an opening.

Definition 2.2 *(BLS Signatures [10])*: The modified BLS signature scheme consists of algorithms (KeyGen, Sign, Verify, Agg, AggVerify):

- $(\mathtt{vk}, \mathtt{sk}) \leftarrow \text{KeyGen}(1^\lambda)$: Generates a verification key $\mathtt{vk} = g_2^x$ and signing key $\mathtt{sk} = x$.
- $\sigma \leftarrow \text{Sign}(\mathtt{sk}, T)$: Signs a message T as $\sigma = H(T)^{\mathtt{sk}}$.
- $b \leftarrow \text{Verify}(\mathtt{vk}, T, \sigma)$: Verifies a signature.
- $\sigma \leftarrow \text{Agg}((\sigma_1, ..., \sigma_k), (\mathtt{vk}_1, ..., \mathtt{vk}_k))$: Aggregates signatures using Lagrange interpolation.
- $b \leftarrow \text{AggVerify}(\sigma, (\mathtt{vk}_1, ..., \mathtt{vk}_k), (T_1, ..., T_k))$: Verifies an aggregate signature.

2.2 Signature-Based Witness Encryption

Definition 2.3 *(Signature-Based Witness Encryption [11])*: A t-out-of-n swe for an aggregate signature scheme (KeyGen, Sign, Verify, Agg, AggVerify, Prove, Valid) is a tuple of two algorithms (Enc, Dec) where:

- $\mathtt{ct} \leftarrow \text{Enc}\Big(1^\lambda, V = (\mathtt{vk}_1, ..., \mathtt{vk}_n), \{T_i\}_{i\in[\ell]}, \{m_i\}_{i\in[\ell]}\Big)$: Encryption takes as input a set V of n verification keys of the underlying scheme Sig, a list of reference signing messages T_i and a list of messages m_i of arbitrary length $\ell \in \text{poly}(\lambda)$. It outputs a ciphertext $\mathtt{ct}$.
- $m \leftarrow \text{Dec}\Big(\mathtt{ct}, \{\sigma_i\}_{i\in[\ell]}, U, V\Big)$: Decryption takes as input a ciphertext $\mathtt{ct}$, a list of aggregate signatures $\{\sigma_i\}_{i\in[\ell]}$ and two sets U, V of verification keys of the underlying scheme Sig. It outputs messages $\{m_i\}_{i\in[\ell]}$.

Definition 2.4 *(Robust Correctness, [11])*: A t-out-of-n SWE scheme $\text{SWE} = (\text{Enc}, \text{Dec})$ for an aggregate signature scheme $\text{Sig} = (\text{KeyGen}, \text{Sign}, \text{Verify}, \text{Agg}, \text{AggVerify}, \text{Prove}, \text{Valid})$ is correct if for all $\lambda \in \mathbb{N}$ and $\ell = \text{poly}(\lambda)$ there is no PPT adversary $\mathcal{A}$ with more than negligible probability of outputting an index $\text{ind} \in [\ell]$, a set of keys $V = (\mathtt{vk}_1, ..., \mathtt{vk}_n)$, a subset $U \subset V$ with $|U| \geq t$, message lists $(m_i)_{i\in[\ell]}, (T_i)_{i\in[\ell]}$ and signatures $(\sigma_i)_{i\in[\ell]}$, such that

$$\text{AggVerify}\Big(\sigma_{\text{ind}}, U, (T_{\text{ind}})_{i\in[\![U]\!]}\Big) = 1, \tag{1}$$

but

$$\text{Dec}\Big(\text{Enc}\Big(1^\lambda, V, (T_i)_{i\in[\ell]}, (m_i)_{i\in[\ell]}\Big), (\sigma_i)_{i\in[\ell]}, U, V\Big)_{\text{ind}} \neq m_{\text{ind}}. \tag{2}$$

Definition 2.5 *(Security)*: A t-out-of-n swe scheme $\text{SWE} = (\text{Enc}, \text{Dec})$ for an aggregate signature scheme $\text{Sig} = (\text{KeyGen}, \text{Sign}, \text{Verify}, \text{Agg}, \text{AggVerify}, \text{Prove}, \text{Valid})$ is secure if for all $\lambda \in \mathbb{N}$, such that $t = \text{poly}(\lambda)$, and all $\ell = \text{poly}(\lambda)$, subsets $\text{SC} \subset [\ell]$, there is no PPT adversary $\mathcal{A}$ that has more than negligible advantage in the experiment $\mathsf{Exp}_{\text{SWE}}(\mathcal{A}, 1^\lambda)$. We define $\mathcal{A}$'s advantage by $\mathsf{Adv}^{\mathcal{A}}_{\text{SWE}} = |\Pr[\mathsf{Exp}_{\text{SWE}}(\mathcal{A}, 1^\lambda) = 1] - \frac{1}{2}|$.

Experiment $\mathsf{Exp}_{\text{SWE}}(A, 1^\lambda)$

1. Let $H_{\mathcal{P}r}$ be a fresh hash function from a keyed family of hash functions, available to the experiment and $\mathcal{A}$.
2. The experiment generates $n - t + 1$ key pairs for $i \in \{t, ..., n\}$ as $(\mathtt{vk}_i, \mathtt{sk}_i) \leftarrow \text{Sig.KeyGen}(1^\lambda)$ and provides $\mathtt{vk}_i$ as well as $\text{Sig.Prove}^{H_{\mathcal{P}r}}(\mathtt{vk}_i, \mathtt{sk}_i)$ for $i \in \{t, ..., n\}$ to $\mathcal{A}$.
3. $\mathcal{A}$ inputs $\text{VC} = (\mathtt{vk}_1, ..., \mathtt{vk}_{t-1})$ and $(\pi_1, ..., \pi_{t-1})$. If for any $i \in [t-1]$, $\text{Sig.Valid}(\mathtt{vk}_i, \pi_i) = 0$, we abort. Else, we define $V = (\mathtt{vk}_1, ..., \mathtt{vk}_n)$.

4. $\mathcal{A}$ gets to make signing queries for pairs (i, T). If $i < t$, the experiment aborts, else it returns $\text{Sig.Sign}(\mathtt{sk}_i, T)$.
5. The adversary announces challenge messages m_i^0, m_i^1 for $i \in \text{SC}$, a list of messages $(m_i)_{i\in[\ell]\setminus \text{SC}}$ and a list of signing reference messages $(T_i)_{i\in[\ell]}$. If a signature for a T_i with $i \in \text{SC}$ was previously queried, we abort.
6. The experiment flips a bit $b \leftarrow \{0,1\}$, sets $m_i = m_i^b$ for $i \in \text{SC}$ and sends $\text{Enc}\Big(1^\lambda, V, (T_i)_{i\in[\ell]}, (m_i)_{i\in[\ell]}\Big)$ to $\mathcal{A}$.
7. $\mathcal{A}$ gets to make further signing queries for pairs (i, T). If $i \geq t$ and $T \neq T_i$ for all $i \in \text{SC}$, the experiment returns $\text{Sig.Sign}(\mathtt{sk}_i, T)$, else it aborts.
8. Finally, $\mathcal{A}$ outputs a guess b'.
9. If $b = b'$, the experiment outputs 1, else 0.

2.3 Extractable Witness KEM

Definition 2.6 *(Indexed Family of NP Relations)*: Let $I \subseteq \{0,1\}^*$ be a set. A set $\mathcal{F} = \{R_I\}_{I\in\mathcal{I}}$ is a family of NP relations with index set I if for all $I \in \mathcal{I}$, R_I is an NP relation. We call I the index of R_I and R_I the relation identified by I. We use $\mathcal{L}_I$ to refer to the corresponding NP language.

Definition 2.7 *(Witness Key Encapsulation Mechanism)*: A witness key encapsulation mechanism for a family of NP relations $\mathcal{F}$ and a keyspace $\mathcal{K}$ is a pair of PPT algorithms $\text{WKEM} = (\mathsf{Encap}, \mathsf{Decap})$, defined as follows:

- $(\mathtt{ct}, k) \leftarrow \mathsf{Encap}(I, x)$: The encapsulation algorithm takes as input an index I identifying a relation $\mathcal{R}_I \in \mathcal{F}$ and a statement x and returns as output a ciphertext $\mathtt{ct}$ and a key $k \in K$.
- $k \leftarrow \mathsf{Decap}(I, w, \mathtt{ct})$: The deterministic decapsulation algorithm takes as input an index I identifying a relation $\mathcal{R}_I \in \mathcal{F}$, a witness $*\, w\, *$, and a ciphertext $\mathtt{ct}$ and returns a key $k \in \mathcal{K}$.

Definition 2.8 *(Correctness)*: A witness KEM $\text{WKEM} = (\mathsf{Encap}, \mathsf{Decap})$ for a family of NP relations $\mathcal{F}$ is correct, if for any $\mathcal{R}_I \in \mathcal{F}$, any $\lambda \in \mathbb{N}$, any $(x, w) \in \mathcal{R}_I$, and any $(\mathtt{ct}, k) \leftarrow \mathsf{Encap}(I, x)$ it holds that $\mathsf{Decap}(I, w, \mathtt{ct}) = k$.

Definition 2.9 *(Extractability)*: A witness KEM $\text{WKEM} = (\mathsf{Encap}, \mathsf{Decap})$ for a family of NP-relations $\mathcal{F}$ is extractable, if there exists a PPT algorithm Ext such that for any stateful PPT adversary $\mathcal{A}$ and any relation $\mathcal{R}_I \in \mathcal{F}$ such that

$$\Pr\Big[\mathsf{Exp}^{\text{KEM-CPA}}_{\text{WKEM},\mathcal{A}}(1^\lambda, I) = 1\Big] \geq \frac{1}{2} + \varepsilon(\lambda) \tag{3}$$

for some non-negligible function $\varepsilon(\lambda)$, it holds that

$$\Pr\big[(x, w) \in \mathcal{R}_I : x \leftarrow \mathcal{A}(1^\lambda, I); w \leftarrow \mathsf{Ext}^{\mathcal{A}(\cdot,\cdot)}(I, x)\big] \geq \delta(\lambda), \tag{4}$$

for some non-negligible function $\delta(\lambda)$. The latter probability is taken over the random coins of the adversary and the extractor and the experiment $\mathsf{Exp}^{\text{KEM-CPA}}_{\text{WKEM},\mathcal{A}}(1^\lambda)$ is defined as follows.

Experiment $\mathsf{Exp}^{\text{KEM-CPA}}_{\text{WKEM},\mathcal{A}}(1^\lambda, I)$

1. $x \leftarrow \mathcal{A}(1^\lambda, I)$ for relationship I

2. Sample $b \leftarrow \{0,1\}$
3. The experiment run the encapsulation algorithm to get test key k_0 $(\mathtt{ct}, k_0) \leftarrow \mathsf{Encap}(I, x)$
4. Sample a random key $k_1 \leftarrow \mathcal{K}$
5. $b' \leftarrow \mathcal{A}(\mathtt{ct}, k_b)$
6. Return 1 if $b = b'$ and 0 otherwise.

2.3.1 Witness KEM for KZG Commitments

We can view the work of Ref. [12] as a witness KEM for univariate polynomials where the relationship, I, is identified by a commitment KZG.$\mathtt{comm}$ and a pair $((\alpha, \beta), \pi) \in \mathcal{R}_I$ if and only if KZG.Verify$(\mathtt{ck}, \text{KZG.}\mathtt{comm}, \pi, \alpha, \beta) = 1$.

We formalize the above in the following definition:

Definition 2.10 *(Witness KEM for KZG Commitments)*: A witness KEM for KZG commitments is a tuple of algorithms $(\mathsf{Encap}, \mathsf{Decap})$, where:

- $(\mathtt{ct}, k) \leftarrow \mathsf{Encap}\big(1^\lambda, \mathtt{comm}, (\alpha, \beta)\big)$: The encapsulation algorithm takes as input a KZG commitment, $\mathtt{comm}$, a point α in $\mathbb{F}_p$, and a value β in $\mathbb{F}_p$ and returns as output a ciphertext $\mathtt{ct}$.
- $k \leftarrow \mathsf{Decap}(\mathtt{comm}, w, \mathtt{ct})$: The decapsulation algorithm takes as input relationship $\mathtt{comm}$ identifying a relation $\mathcal{R}_{\mathtt{comm}}$, a witness w, and a ciphertext $\mathtt{ct}$ and returns a key k.

2.4 Garbled Circuits

Similar to Ref. [2], we use garbled circuits as defined by Ref. [14] though we modify the definition to allow for multi-bit outputs.

Definition 2.11 *(Garbled Circuit, [14,15])*: Let $\{\mathcal{C}_n\}_n$ be a family of circuits where each circuit in $\mathcal{C}_n$ has n bit inputs. A garbling scheme, $\mathtt{GC}$, for circuit family $\{\mathcal{C}_n\}_n$ consists of polynomial-time algorithms $(\mathtt{Garble}, \mathtt{Eval})$:

- $\mathtt{Garble}\big(1^\lambda, C \in \mathcal{C}_n\big)$: The algorithm takes a security parameter λ and a circuit C as input and outputs a garbled circuit G together with $2n$ wire keys $\big\{w_{i,b}\big\}_{i\in[n],b\in\{0,1\}}$
- $\mathtt{Eval}\Big(G, \{w_i\}_{i\in[n]}\Big)$: The evaluation algorithm takes as input a garbled circuit G and n wire keys $\{w_i\}_{i\in[n]}$ and outputs $y \in \{0,1\}^m$.

Definition 2.12 *(Garbling Correctness)*: A garbling scheme $\mathtt{GC}$ for circuit family $\{\mathcal{C}_n\}_n$ is said to be correct if for all λ, n, $x \in \{0,1\}^n$ and $C \in \mathcal{C}_n$,

$$\mathtt{Eval}\Big(G, \big\{w_{i,x_i}\big\}(i \in [n])\Big) = C(x), \tag{5}$$

where

$$\Big(G, \big\{w_{i,b}\big\}_{i\in[n],b\in\{0,1\}}\Big) \leftarrow \mathtt{Garble}\big(1^\lambda, C\big). \tag{6}$$

Definition 2.13 *(Selective Security)*: A garbling scheme $\mathtt{GC} = (\mathtt{Garble}, \mathtt{Eval})$ for a class of circuits $\mathcal{C} = \{\mathcal{C}_n\}_n$ is said to be a selectively secure garbling scheme if there exists a polynomial-time simulator Sim such that for all λ, n, $C \in \mathcal{C}_n$ and $x \in \{0,1\}^n$, the following holds:

$$\left\{\mathrm{Sim}\left(1^\lambda, 1^n, 1^{|C|}, C(x)\right)\right\} \overset{c}{\approx} \left\{ \left(G, \left\{w_{i,x_i}\right\}_{i\in[n]}\right) : \left(G, \left\{w_{i,b}\right\}_{i\in[n],b\in\{0,1\}}\right) \leftarrow \mathtt{Garble}\left(1^\lambda, C\right) \right\} \tag{7}$$

We also require that if *no* wire keys are given, then the garbled circuit is indistinguishable from a random string of the same length.

Definition 2.14 *(Indistinguishability)*: A garbling scheme $\mathtt{GC} = (\mathtt{Garble}, \mathtt{Eval})$ for a class of circuits $\mathcal{C} = \{\mathcal{C}_n\}_n$ is said to be indistinguishable from random if for all λ, n, $C \in \mathcal{C}_n$, the following holds:

$$\left\{\mathrm{Sim}\left(1^\lambda, 1^n, 1^{|C|}\right)\right\} \overset{c}{\approx} \left\{G : \left(G, \left\{w_{i,b}\right\}_{i\in[n],b\in\{0,1\}}\right) \leftarrow \mathtt{Garble}\left(1^\lambda, C\right)\right\}. \tag{8}$$

3 Blockchain Formalization

Rather than defining a fully fledged blockchain, we will define a more abstract and general notion of a "signature-based blockchain" (which we will denote $\mathcal{BC}$). All a signature-based blockchain needs to do is have a set of verification keys, a way to define "consensus" based on some threshold t of the signatures, and a way to define the state of the blockchain.

Definition 3.1 *(Blockchain Template)*: We define a blockchain as a tuple $\mathcal{BC} = (\mathrm{Verify}, \mathtt{state})$ where $\mathtt{state}$ is a mutable state of the blockchain and Verify is a *fixed* verification algorithm. Further, we have the following verification algorithm:

$$\mathrm{Verify}(\mathtt{state}') = 1 \tag{9}$$

if and only if $\mathtt{state}'$ is a valid state of the blockchain.

3.1 Signature Based Blockchain

We now narrow down the definition of a blockchain to a more specific notion of a "signature-based blockchain" off of which we will build. At a high level, a signature-based blockchain is a blockchain that has a set of verification keys, a way to verify signatures on the state of the blockchain, and a way to verify the set of verification keys.

Definition 3.2 *(Signature Based Blockchain)*: We define a signature-based blockchain at time T as a tuple $\mathcal{BC}_T = (\mathrm{Verify}_V, \mathtt{state}^*, t)$ where $\mathtt{state}^* = \mathtt{state}_1, ..., \mathtt{state}_{T-1}$ and $\mathrm{Verify}(\mathtt{state}^* \parallel \mathtt{state}_T) = 1$ if and only if for $\mathtt{vk}_1, ..., \mathtt{vk}_n \leftarrow V(\mathtt{state}_{T-1})$ and $\mathtt{state}_T = \left(\sigma_T, V_{T+1} \parallel \widetilde{\mathtt{state}}_T\right)$ is valid if and only if σ_T is a valid signature for $V_{T+1} \parallel \widetilde{\mathtt{state}}_T$ under at least t verification keys in $\mathtt{vk}_1, ..., \mathtt{vk}_t \subseteq V_T$. We also have that $\mathrm{Verify}(\mathtt{state}_1) = 1$ if and only if $\mathtt{state}_1$ is some the fixed initial state of the blockchain.

4 $T+1$-Extractable Witness Encryption for Blockchain

We now present our modification on "extractable witness encryption" for blockchains (eWEBs) [4]. We will simplify the definition as, unlike Ref. [4], we will not need to worry about the underlying commitees in the blockchain. Moreover, we restrict the eWEB to what we term a $T+1$-eWEB: where decryption can only occur if the state of the *next* block[7] has some target state at position $\mathtt{idx}$ in the blockchain.

[7]Here time does not literally refer to time on earth, but rather the block number.

Definition 4.1 *($T+1$-eWEB for Signature-Based Blockchain)*: For the T-th block and signature-based blockchain blockchain $\mathcal{BC}_T = (\text{Verify}_V, \texttt{state}^*, t)$ for $\texttt{state}^* = (\texttt{state}_1, ..., \texttt{state}_T)$, let V_{T+1} be the verification algorithm for the $T+1$-th block. Then, a $T+1$-eWEB is a tuple of algorithms $(\texttt{SecretStore}, \texttt{SecretRelease})$ where:

- $\texttt{SecretStore}(w, V_{T+1}, \texttt{idx}, \texttt{target})$: the sending party encrypts w which can be released if $\texttt{state}_{T+1}[\texttt{idx}] = \texttt{target}$ and $\text{Verify}_{T+1}(\texttt{state}_{T+1}) = 1$. The secret then has an associated identifier, id
- $\texttt{SecretRelease}(\text{id}) \to w$ or $\perp$: a requester can request the secret w associated with id if $\texttt{state}_{T+1}[\texttt{idx}] = \texttt{target}$ is true valid state $\texttt{state}_{T+1}$. If $\texttt{state}_{T+1}[\texttt{idx}] = \texttt{target}$ is not true, then the requester will get "bot" (or some other value) as the output.

We define the security of $T+1$-eWEBs relative to the security of the underlying blockchain: i.e. only if we can produce two distinct valid states for the same block (and thus breaking the blockchain), then we can break the security of the $T+1$-eWEB. Unfortunately, we are unable to give a "clean" and general definition of the security of $T+1$-eWEBs due to our scheme's construction in Section 4.1. Thus, we defer the formal definition to Section 4.1 where we will give a concrete instantiation of a time-based eWEB.

4.1 Instantiating Time-Based eWEBs

We now show how to instantiate time-based eWEBs via a series of circuits and smart-contracts. First though, we will need to define our underlying blockchain model.

4.1.1 Signature Based Blockchain Instantiation

Before we can fully construct our time-based eWEB, we need to make a few concrete "design decisions" about the underlying blockchain. We note that the blockchain itself can be constructed in a variety of ways, but for our purpose we need three main properties:

1. KZG polynomial commitments to the state at each time step
2. BLS signatures for the verification keys and signing of the root (though with a slight modification as in Ref. [11]).
3. And **bit-by-bit** signatures of the KZG root. Rather than directly signing the KZG commitment, `comm`, we will require a separate BLS signature for each **bit** of the KZG commitment when represented in binary.

We note that the first two properties are quite close to the existing Ethereum blockchain, but the third property is not.

We sketch out what the blockchain verification algorithm looks like in Blockchain 1.

Blockchain 1: Sketch of a Signature Based Blockchain. We do not specify a consensus algorithm

State $\texttt{state}^* = (\texttt{state}_1, ..., \texttt{state}_T)$ for the first T blocks where for blockchain data $\widetilde{\texttt{state}}_T$,

$$\texttt{state}_T = \big(\sigma_T, V_{T+1}, \widetilde{\texttt{state}}_T\big) \tag{10}$$

where

$$\sigma_j = \text{Agg}\Big(\big(\sigma_j^i\big)_{i\in U}, (\texttt{vk}_i)_{i\in U}\Big) \tag{11}$$

and $\sigma_j^i = \text{Sign}\big(\texttt{sk}_i, P_j\big)$ for

$$P_j = H\left(T \parallel j \parallel \underbrace{0...0}_{j-1}...b_j... \underbrace{0...0}_{\lambda_{\mathbb{G}}-j-1}\right) \tag{12}$$

and

$$b_j = \text{Commit}\big(\texttt{ck}, V_{T+1} \parallel \widetilde{\texttt{state}}_T\big)[j], \text{ for } \ j \in [\lambda_{\mathbb{G}}] \tag{13}$$

$\text{Verify}(\texttt{state}^*) = 1$ if and only if:

For all $i \in 2, ..., T$

Let $V_i = (\texttt{vk}_1, ..., \texttt{vk}_n)$ be defined by $\texttt{state}_{i-1}$, $U_T \subseteq V_T$ and $|U_T| \geq t$, $\texttt{state}_T = \Big(\sigma_T = \big(\sigma_{T,1}, ...\sigma_{T,\lambda_{\mathbb{G}}}\big), V_{T+1}, \widetilde{\texttt{state}}_T\Big)$ is valid if and only if

$$\text{AggVerify}\Big(\sigma_{T,j}, (\texttt{vk}_i)_{i\in U_T}, P_j\Big) = 1 \tag{14}$$

for all $j \in [\lambda_{\mathbb{G}}]$

And $\text{Verify}(\texttt{state}_1) = 1$ (i.e. the genesis block is publicly known and acknowledged)

As we can see in Blockchain 1, the verification algorithm is quite simple and *almost* standard except for the bit-by-bit signatures.

Now that our blockchain is defined, we define our security definition for the $T+1$-eWEBs. Specifically, the security will be defined relative to the security of the underlying blockchain where we will say that the blockchain is broken if the adversary can produce two *distinct* though valid signatures for the j-th bit of the KZG commitment. As in most proof of stake blockchains, we can still have a slashing condition which will allow us to "punish" any validators which produce two distinct signatures for the same bit of the KZG commitment.

Definition 4.2 *(Security of $T+1$-eWEB)*: Let signature-based $\mathcal{BC} = (\text{Verify}_V, \texttt{state}^*, t)$ be a blockchain with $V_{T+1} = (\texttt{vk}_1, ..., \texttt{vk}_n)$ being the verification keys for time $T+1$ and $\texttt{state}$ be the state of the blockchain. Let $\mathcal{BC}_0$ be the initial state of the blockchain and $\mathcal{BC}_T$ be the state of blockchain immediately prior to calling $\texttt{SecretStore}(s, \mathcal{BC}_T, \texttt{idx}, \texttt{target})$. We say that an $T+1$-eWEB is secure if for any BPP adversary $\mathcal{A}$ which can corrupt up to $t-1$ parties at any time, then the following holds: If for any $s, s', T, \texttt{idx}, \texttt{target}$,

$$\begin{aligned}\big|\Pr\big[b = 1 \mid b \leftarrow \mathcal{A}\big(1^\lambda, \mathcal{BC}, \texttt{SecretStore}(s, \mathcal{BC}_{\textbf{Init}}, T, \texttt{idx}, \texttt{target})\big)\big] - \\ \Pr\big[b = 1 \mid b \leftarrow \mathcal{A}\big(1^\lambda, \mathcal{BC}, \texttt{SecretStore}(s', \mathcal{BC}_{\textbf{Init}}, T, \texttt{idx}, \texttt{target})\big)\big]\big| > \text{negl}(\lambda)\end{aligned} \tag{15}$$

then there exists a BPP extractor E such that for the check in eq. (14) with output $C_j \in \{0,1\}$ on $\texttt{state}^*$ and $\overline{C_j}$ on $\overline{\texttt{state}^*}$, and some $j \in [\lambda_{\mathbb{G}}]$:

$$\Pr\Big[\mathtt{comm}_j \neq \overline{\mathtt{comm}}_j \ \text{ and } \ C_j = 1, \overline{C_j} = 1 \\ \text{or } \ \mathrm{Verify}_V(\mathtt{state}^*) = 1 \ \text{ and } \ \mathtt{state}_{T+1}[\mathtt{idx}] = \mathtt{target} \ \Big| \\ \mathtt{state}^*, \overline{\mathtt{state}^*} \leftarrow E\big(1^\lambda, \mathcal{BC}\big)\Big] \geq \mathrm{poly}^{-1}(\lambda). \tag{16}$$

In words, our soundness says that if the adversary can break semantic security of the secret s without the condition being met, then the adversary can produce two distinct signatures for the same bit of the KZG commitment, breaking the blockchain[8].

4.1.2 eWEB Scheme

We are now ready to present our main technical contribution: our $T+1$-eWEB scheme, which will be the building block for the rest of the paper. Protocol 2 is our $T+1$-eWEB scheme which is based on the signature-based blockchain with verification procedure along the lines of Blockchain 1.

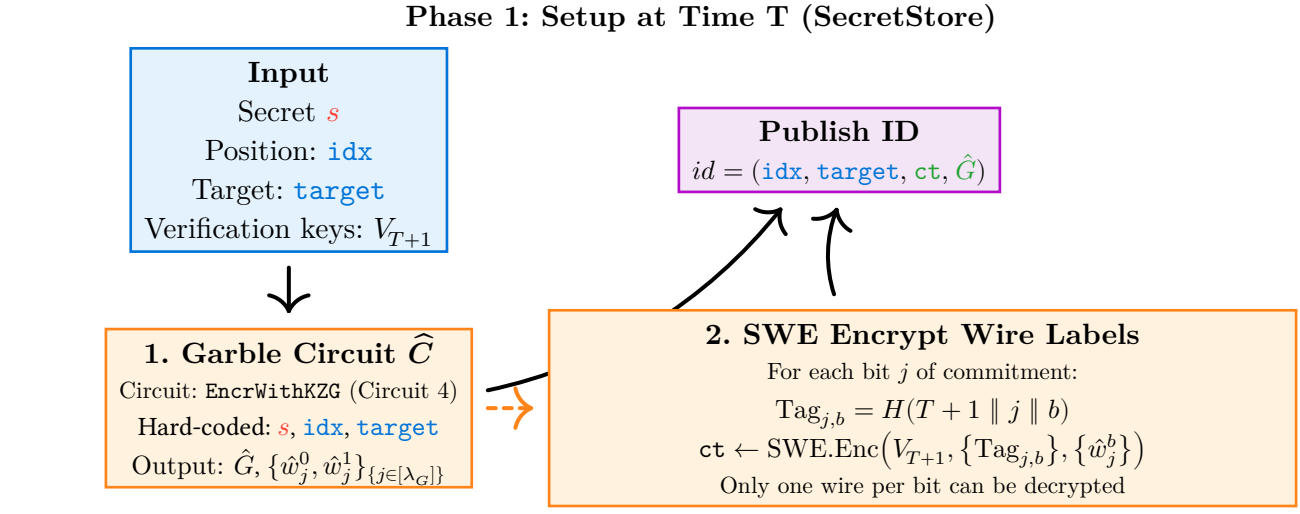

Figure 6: A visualization of the setup phase for our $T+1$-eWEB scheme.

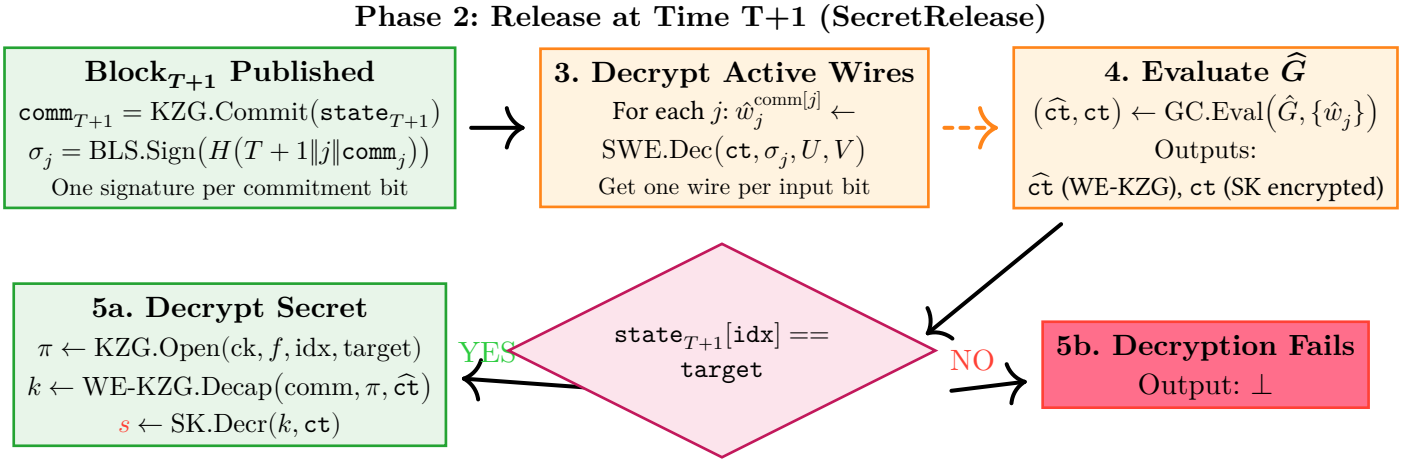

Figure 7: A visualization of the release phase for our $T+1$-eWEB scheme.

[8]Why can we not define the security in terms of an adversary producing two valid states? We use garbled circuits where security doesn't hold if the wire values for $0, 1$ for the same input bit are known to the adversary. As we shall see, if an adversary can produce two valid $\mathtt{comm}_j, \overline{\mathtt{comm}_j}$ with $\mathtt{comm}_j \neq \overline{\mathtt{comm}_j}$, then the adversary can decrypt two different wires for the same input bit, without necessarily having to come up with a fully valid blockchain state.

We provide a block-diagram visualization of the scheme in Figure 6 and Figure 7. Now, we will describe the formal details in Protocol 2 alongside a helper algorith, Algorithm 3, and a fixed circuit, Circuit 4.

Protocol 2: $T+1$-eWEB

$\texttt{SecretStore}(s, V_{T+1}, \texttt{idx}, \texttt{target})$ for message s and $V_{T+1} = (\texttt{vk}_1, ..., \texttt{vk}_n)$:

1 Garble the circuit $\hat{C}$ defined in Circuit 4 with hard-coded s, data position, $\texttt{idx}$, and target data, $\texttt{target}$. The labels are $\hat{w}_j^0, \hat{w}_j^1$ for $j \in [\lambda_{\mathbb{G}}]$ and the garbled circuit is $\hat{G}$.

2 Set

$$\texttt{ct} \leftarrow \text{SWE.Enc}\left(1^\lambda, V_{T+1}, \left\{T+1 \parallel j \parallel \underbrace{0...0}_{j-1} ... b ... \underbrace{0...0}_{\lambda_{\mathbb{G}}-j-1}\right\}_{b\in\{0,1\},j\in[\lambda_{\mathbb{G}}]}, \left\{\hat{w}_j^b\right\}_{b\in\{0,1\},j\in[\lambda_{\mathbb{G}}]}\right) \tag{17}$$

3 Publish $\text{id} = \left(\texttt{idx}, \texttt{target}, \texttt{ct}, \hat{G}\right)$

$\texttt{SecretRelease}\left(\text{id} = \left(\texttt{idx}, \texttt{target}, \texttt{ct}, \hat{G}\right)\right)$ for blockchain $\mathcal{BC}_{T+1}$:

4 For time $T+1$, let $U_{T+1} \subseteq V_{T+1}$ be the signing set and $V_{T+1} = (\texttt{vk}_1, ..., \texttt{vk}_n)$ be the verification keys.

5 If $|U_{T+1}| < t$

6 | **Return** $\perp$.

7 Call $\hat{w}_1^{\texttt{comm}_1}, ..., \hat{w}_{\lambda_{\mathbb{G}}}^{\texttt{comm}_{\lambda_{\mathbb{G}}}} = \texttt{DecrWires}(\sigma, V, U, \texttt{ct}, \texttt{comm})$ (Algorithm 3)

8 Let $\widehat{\texttt{ct}}, \texttt{ct} = \texttt{GC.Eval}\left(\hat{G}, \left\{\hat{w}_i^{\texttt{comm}_i}\right\}_{i\in[\lambda_{\mathbb{G}}]}\right)$ Use the wire labels $\hat{w}_i^{\texttt{comm}_i}$ to evaluate Circuit 4 to get $\widehat{\texttt{ct}}, \texttt{ct}$

9 Set $\pi = \text{KZG.Open}(\texttt{ck}, f, \texttt{idx}, \texttt{target})$

10 Set $k = \mathsf{Decap}\left(\texttt{comm}, \pi, \widehat{\texttt{ct}}\right)$

11 **Return** $s = \texttt{sk.Decr}(k, \texttt{ct})$

Algorithm 3: $\texttt{DecrWires}(\sigma, V, U, \texttt{ct}, \texttt{comm})$, decrypt wire-labels for the first-stage garbled-circuit

---

Input: Signatures σ_i for $i \in [\lambda_{\mathbb{G}}]$, verification keys $V = (\texttt{vk}_1, ..., \texttt{vk}_n)$, signing set $U \subseteq [n]$, ciphertexts $\texttt{ct}$, and state root commitment $c = \texttt{comm}$.

1 **For** $i \in [\lambda_{\mathbb{G}}]$,

2 | Set $\hat{w}_i^{c_i} = \text{SWE.Dec}(\texttt{ct}, \sigma_i, U, V)$

3 **Return** wire labels $\hat{w}_i^{c_i}$ for $i \in \left[\lambda_{\text{garb}}\right]$.

Circuit 4: `EncrWithKZG(comm)`, first-stage circuit to be garbled

Input: Commitment to KZG polynomial `comm`

Hard-Coded Values: Message s and fixed data-positions, `idx`, and target `target`

1 For $b \in \{0,1\}$, let $\widehat{\mathtt{ct}}, k \leftarrow \mathsf{Encap}\big(1^\lambda, \mathtt{comm}, (\mathtt{idx}, \mathtt{target})\big)$ as in Definition 2.10

2 Set $\mathtt{ct} = \mathtt{sk}.\mathtt{Encr}(k, s)$

3 **Return** $\widehat{\mathtt{ct}}, \mathtt{ct}$

Theorem 4.1 (Correctness): Protocol 2 is correct (Definition 4.1) assuming that the signature-based witness encryption is correct, and the KZG base witness encryption is correct.

Proof: The proof follows from direct inspection of the protocol. If $\mathtt{state}_{T+1}[\mathtt{idx}] = \mathtt{target}$, then the blockchain will produce a valid signature σ_j for the j-th bit of the KZG commitment, `comm`, for all $j \in [\lambda_{\mathbb{G}}]$. Then, we can decrypt the wire labels $\hat{w}_j^{\mathtt{comm}_j}$ for all $j \in [\lambda_{\mathbb{G}}]$ using the signature-based witness encryption correctness (Definition 2.4). By the simulation soundness of the garbled circuits (Definition 2.13), we can evaluate the garbled circuit $\hat{G}$ on the wire labels $\hat{w}_j^{\mathtt{comm}_j}$ to get $\widehat{\mathtt{ct}}, \mathtt{ct}$. Finally, we can use the KZG base witness encryption correctness (Definition 2.8) to decrypt $\widehat{\mathtt{ct}}$ to get the message k and then decrypt `ct` using the secret key k to get the message s. ■

Theorem 4.2 (Soundness): Protocol 2 is sound (Definition 4.2) assuming that the signature-based witness encryption is sound, and the KZG base witness encryption is sound.

We provide a full proof of the above theorem in Appendix A and sketch the proof here to provide intuition.

Proof sketch: The proof proceeds via contrapositive. Assume that there is no PPT extractor, E, which can produce two distinct signatures for the same bit of the KZG commitment for all $j \in [\lambda_{\mathbb{G}}]$ and that E cannot produce $\mathtt{state}^*$ which is a valid blockchain state and meets the $T+1$-eWEB condition.

First, note that for all $j \in [\lambda_{\mathbb{G}}]$, either $\hat{w}_i^0$ or $\hat{w}_i^1$ is indistinguishable from random by the security of signature-based witness encryption (Definition 2.5) as the extractor cannot produce two valid signatures for the same bit of the KZG commitment. And so, we use the selective security of the garbled circuits (Definition 2.13) to replace the garbled circuit with the simulator which allows for evaluating C' at most once and learning nothing else about C'. And so, we can evaluate C' *only* on KZG commitment $\mathtt{comm}_{\mathtt{state}_{T+1}}$ as only the valid signatures for one KZG commitment are revealed. Next, note that the output of C', $\widehat{\mathtt{ct}}, \mathtt{ct}$ are indistinguishable from random by the security of the extractable witness KEM for KZG commitments (Definition 2.9) as $\mathtt{state}_{T+1}[\mathtt{idx}] \neq \mathtt{target}$ and, by the binding

property of the KZG commitment, no proof π can be produced for $\texttt{state}_{T+1}[\texttt{idx}] = \texttt{target}$ in polynomial time. ■

5 Instantiating One-Time Programs

We now show why $T+1$-eWEBs, combined with a smart-contract system, are so powerful. We first start by instantiating $T+1$ one-time programs which allow for a single evaluation of a program by posting a message to the blockchain prior to the $T+1$-th block. We then show how $T+1$-OTPs can be bootstrapped to extremely powerful primitives such as RAM obfuscation. We also include "conditional gating" for the one-time programs which allows us to instantiate "pay-to-use" primitives and other powerful primitives. This section draws heavily from the ideas of Goyal and Goyal [2].

Definition 5.1 *($T+1$-conditional One-Time Program ($T+1$-OTP))*: A $T+1$-conditional one-time program, $T+1$-OTP, is a tuple $(\textbf{Setup}, \textbf{SetInput}, \textbf{Eval})$ where:

- $\textbf{Setup}(\mathcal{BC}_\ell, \texttt{Cond}, r)$ for $\ell \leq T$ is a setup algorithm that takes as input a blockchain at time ℓ, circuit input size r, and a condition Cond and outputs a setup identifier, $\text{id}_{\textbf{Setup}}$.
- $\textbf{Init}\big(V_{T+1}, C, \text{id}_{\textbf{Setup}}\big)$ is an initialization algorithm that outputs an identifier, $\text{id}_{\textbf{Init}}$
- $\textbf{SetInput}\big(\mathcal{BC}_T, \text{id}_{\textbf{Setup}}, x\big)$ is an algorithm that takes as input a value x and sets the input of the OTP to x by updating the blockchain at time T.
- $\textbf{Eval}(\mathcal{BC}_{T+1}, \text{id}_{\textbf{Init}})$ is an algorithm that takes as input the blockchain at time $T+1$ and attempts to output $C(x)$.

Definition 5.2 *(OTP Simulation-Based Soundness)*: We say that a OTP scheme is sound for circuit C relative and auxiliary information, aux, if:

- For every adversary, $\mathcal{A}$, there exists a simulator Sim such the following holds:

$$\begin{aligned}\mathcal{A}\big(1^\lambda, 1^{|C|}, \mathcal{BC}_{T+1}, \text{id}_{\textbf{Setup}} \leftarrow \textbf{Setup}(\mathcal{BC}_\ell, \texttt{Cond}), \texttt{Cond}, \textbf{Init}\big(V_{T+1}, C, \text{id}_{\textbf{Setup}}\big)\big)\\ \overset{c}{\approx} \mathsf{Sim}^{\texttt{Single}(C)}\big(1^\lambda, 1^{|C|}, \texttt{Cond}, \textbf{Setup}(\mathcal{BC}_\ell, \texttt{Cond}), \mathcal{BC}_{T+1}, \texttt{aux}\big)\end{aligned} \tag{18}$$

where $\texttt{Single}(C)$ allows for only one evaluation of C if and only if $\texttt{Cond}\big(\texttt{state}_{T+1}\big) = 1$.

Notice that the simulator Sim also receives the setup output. Given that the setup only contains publically known information, this is not a problem.

We leave a UC definition of $T+1$-conditional one-time programs for future work.

5.1 Constructing $T+1$-OTPs from $T+1$-eWEB

We now show how to instantiate $T+1$ one-time programs from $T+1$ eWEBs, assuming the existence of a smart contract system.

Smart-Contract 5: Smart Contract for One-Time Program

Hard-Coded values: State with position $\texttt{idx}_1, ..., \texttt{idx}_r$ for functions with r input bits and condition $\texttt{Cond}(\mathcal{BC})$.

Initialization:

Set $\mathcal{BC}[\texttt{idx}_i] = -1$ for all $i \in [r]$

SetInput(x) for $x \in \{0,1\}^r$:

If $\mathcal{BC}[\texttt{idx}_i] \neq -1$ for any i, then
  Return "Error: Input already set"
If $\texttt{Cond}(\mathcal{BC}) = 1$, then
  Set $\mathcal{BC}[\texttt{idx}_i] = x_i$ for all $i \in [r]$
If $\texttt{Cond}(\mathcal{BC}) = 0$, then
  Return "Error: Condition not met"

Protocol 6: $T+1$-eWEB

Setup$(\mathcal{BC}_\ell, \texttt{Cond})$:

Post smart contract $\mathcal{SC}$ as defined in Smart-Contract 5 to the blockchain at time ℓ
Let $\text{id}_{\textbf{Setup}} = (\texttt{idx}_1, ..., \texttt{idx}_r, \texttt{addr})$ be the identifier of the smart contract and $\texttt{addr}$ be the address of the smart contract.
Return $\text{id}_{\textbf{Setup}}$

Init$\big(V_{T+1}, C, \text{id}_{\textbf{Setup}} = (\texttt{idx}_1, ..., \texttt{idx}_r, \texttt{addr})\big)$:

Garble circuit C with r input bits to wires $w_1^0, w_1^1 ..., w_r^0, w_r^1$ to get garbled circuit G.
For $i \in [r], b \in \{0,1\}$
  Let $\text{id}_i^b \leftarrow \texttt{SecretStore}\big(w_i^b, V_{T+1}, \texttt{idx}_i, b\big)$
Return $\text{id}_{\textbf{Init}} = \Big(G, \big\{\text{id}_i^b\big\}_{i\in[r], b\in\{0,1\}}\Big)$

SetInput$\big(\mathcal{BC}_T, \text{id}_{\textbf{Setup}}, x\big)$:

Call **SetInput** on $\mathcal{SC}$ at address $\texttt{addr}$ with the input x
If $\mathcal{SC}$ returns an error
  Return the same error

Eval$\Big(\mathcal{BC}_{T+1}, \text{id}_{\textbf{Init}} = \Big(G, \big\{\text{id}_i^b\big\}_{i\in[r], b\in\{0,1\}}\Big)\Big)$:

For $i \in [r]$
  Let $w_i^{x_i} \leftarrow \texttt{SecretRelease}\big(\text{id}_i^{x_i}\big)$
Call $y = \texttt{GC}.\textbf{Eval}\Big(G, \big\{w_i^{x_i}\big\}_{i\in[r]}\Big)$
Return y

Intuitively, we can see the above protocol as "enforcing" one-timeness by using a smart contract to ensure that the input is only set once.

Theorem 5.1 (One-Time Program Soundness from $T+1$-eWEB): Assuming the soundness of our $T+1$-eWEBs and that only one blockchain state can be produced per block, the protocol above is a sound $T+1$-conditional one-time program, $T+1$-OTP, where soundness is defined as in Definition 5.2.

We give a proof sketch outline below and a more complete proof in Appendix B.

Proof sketch: The proof follows from a simple inspection of the smart contract alongside the security properties of the eWEB. Note that it, for any $i \in [r]$, $\mathtt{state}_{T+1}[\mathtt{idx}_r] \in \{-1, 0, 1\}$ and can only be set to *one* of these values by the soundness of the blockchain. Moreover, for all i, $\mathtt{state}_{T+1}[\mathtt{idx}_i] \neq -1$ if and only if $\mathtt{Cond}(\mathtt{state}_{T+1}) = 1$. And so, if $\mathtt{Cond}(\mathtt{state}_{T+1}) \neq 1$, then all the wires to the garbled circuit are indistinguishable from random by the security of the eWEB (Definition 4.2). Thus, if $\mathtt{Cond}(\mathtt{state}_{T+1}) \neq 1$, the simulator can simply simulate the garbled circuit (by the indistinguishability of garbled circuits, Definition 2.14.

If $\mathtt{Cond}(\mathtt{state}_{T+1}) = 1$ and $\mathtt{state}_{T+1}[\mathtt{idx}_i] \neq -1$, then an adversary can only learn the wires $w_i^{\mathtt{state}_{T+1}[\mathtt{idx}_i]}$ for $i \in [r]$ by the soundness of the $T+1$-eWEB. We then use the soundness of the garbled circuit to show that the simulator can simulate the garbled circuit and its wires while only learning $C(\mathtt{state}_{T+1}[\mathtt{idx}_1] \parallel \mathtt{state}_{T+1}[\mathtt{idx}_2] \parallel ... \parallel \mathtt{state}_{T+1}[\mathtt{idx}_r])$. ■

5.2 Modification to our $T+1$-OTP Construction

Note that in the above, if the smart contract is not updated prior to the $T+1$-th block, then we can never evaluate the garbled circuit on any input. Thus, we will make modifications to our construction to allow for the garbled circuit to be evaluated on $\perp$ if the smart contract was not updated. We will later see, in Section 6, how this can be used to construct RAM obfuscation.

Our modifications are as follows; we need the one-time programs to do the following:
1. Have a "global" setup which only needs to be called once for a series of successive $T+1$-OTPs.
2. Allow for the one-time program to take in the next set of verification keys.
3. Require that the evaluation, **Eval**, outputs $C((\mathtt{vk}_1, ..., \mathtt{vk}_n), \perp)$ if $\mathtt{Cond}(\mathtt{state}_{T+1}) = 0$ or if **SetInput** *was not* called at time T.

We thus modify our $T+1$-OTP as follows:

Definition 5.3 *(Modified $T+1$-conditional One-Time Program ($T+1$-OTP'))*: A modified $T+1$-conditional one-time program, $T+1$-OTP', is a tuple $(\mathbf{GlobalSetup}, \mathbf{SetInput}, \mathbf{Eval})$ where:
- $\mathbf{GlobalSetup}(\mathcal{BC}_\ell, \mathtt{Cond}, r)$ for $\ell < T$ is a setup algorithm that takes as input a blockchain at time ℓ, circuit input size r, and a condition $\mathtt{Cond}$ and outputs a setup identifier, $(\mathrm{id}_{\mathbf{Setup}}, T_{\mathbf{Init}})$.
- $\mathbf{Init}(V_{T+1}, C, \mathrm{id}_{\mathbf{Setup}}, T_{\mathbf{Init}})$ is an initialization algorithm that outputs an identifier, $\mathrm{id}_{\mathbf{Init}}$ for $C : \mathcal{V} \times \{\perp\} \cup \{0,1\}^r \to \mathcal{Y}$ where $\mathcal{V}$ is the domain of possible verification keys. $T_{\mathbf{Init}}$ can be thought of as the last time in which the smart-contract, $\mathcal{SC}$, was updated.

- **SetInput**$\big(\mathcal{BC}_T, \mathrm{id}_{\mathbf{Setup}}, x\big)$ is an algorithm that takes as input a value x and sets the input of the OTP to x by updating the blockchain at time T.
- **Eval**$(\mathcal{BC}_{T+1}, \mathrm{id}_{\mathbf{Init}})$ is an algorithm that takes as input the blockchain at time $T+1$ and attempts to output $C((\mathtt{vk}_1, ..., \mathtt{vk}_n), x)$ where $\mathtt{vk}_1, ..., \mathtt{vk}_n$ are the verification keys for $\mathtt{state}_{T+2}$. If $\mathtt{Cond}\big(\mathtt{state}_{T+1}\big) = 0$ or if **SetInput** was not called at time T, then **Eval** outputs $C((\mathtt{vk}_1, ..., \mathtt{vk}_n), \bot)$.

To accommodate the above change, our construction remains almost the same except that we:

1. Need to have a bit-by-bit encoding of the next set of verification keys within the smart contract. Specifically, we need to change eq. (13) in our blockchain specification to:
$$b_j = \mathrm{Commit}\big(\mathtt{ck}, \mathtt{Bitify}(V_{T+1}) \parallel \widetilde{\mathtt{state}}_T\big)[j], \text{ for } \ j \in [\lambda_{\mathbb{G}}] \tag{19}$$
where $\mathtt{Bitify}$ is a function which takes as input a string and outputs the bit-by-bit encoding of the string[9].
2. Need to have a "last-called" slot for the smart contract which is set to T when **SetInput** is called at time T and ℓ at initialization block $\ell < T$
3. Allows for the garbled circuit to be evaluated on $\bot$ if the last-called slot is ℓ and otherwise on x if **SetInput** was called at time T.

We will also slightly modify the security definition:

Definition 5.4 *(*OTP' *Simulation-Based Soundness)*: We say that a OTP scheme is sound for circuit C relative and auxiliary information, $\mathtt{aux}$, if:

- For every adversary, $\mathcal{A}$, there exists a simulator Sim such the following holds:
$$\begin{aligned}\mathcal{A}\big(1^\lambda, 1^{|C|}, \mathcal{BC}_{T+1}, \mathrm{id}_{\mathbf{Setup}} \leftarrow \mathbf{Setup}(\mathcal{BC}_\ell, \mathtt{Cond}), \mathtt{Cond}, \mathbf{Init}\big(V_{T+1}, C, \mathrm{id}_{\mathbf{Setup}}\big)\big)\\ \overset{c}{\approx} \mathsf{Sim}^{\mathtt{Single}(C)}\big(1^\lambda, 1^{|C|}, \mathtt{Cond}, \mathbf{Setup}(\mathcal{BC}_\ell, \mathtt{Cond}), \mathcal{BC}_{T+1}\big)\end{aligned} \tag{20}$$
where $\mathtt{Single}(C)$ allows for only one evaluation of C. If $\mathtt{Cond}\big(\mathtt{state}_{T+1}\big) = 0$, then C can only evaluate on $\bot$. Otherwise, C can evaluate on x or $\bot$ if and only if $\mathtt{Cond}\big(\mathtt{state}_{T+1}\big) = 1$.

Notice that the key difference between the above definition and Definition 5.2 is that the condition on when C can be evaluated is depends on whether the input is $\bot$ or not.

Instantiating $T+1$-OTP'

We now show how to instantiate $T+1$-OTP' from a smart contract system. The general outline is similar to Protocol 6 but we need to add a few more details.

[9]Note that the change to Equation 13 increases the state of the blockchain by a multiplicative factor of $O(\lambda_{\mathbb{G}})$ bits which can be substantial. We leave the optimization of this to future work.

Smart-Contract 7: Smart Contract for One-Time Program

Hard-Coded values: State with position $\texttt{idx}_1, ..., \texttt{idx}_r$ for functions with r input bits and condition $\texttt{Cond}(\mathcal{BC})$.

Initialization at block ℓ:

1 Set $\mathcal{BC}[\texttt{idx}_i] = 0$ for all $i \in [r]$

2 Set $\texttt{last\_update} = \ell$

SetInput(x) for $x \in \{0,1\}^r$ called at time T:

3 **If** $\texttt{Cond}(\mathcal{BC}) = 1$, then

4 Set $\mathcal{BC}[\texttt{idx}_i] = x_i$ for all $i \in [r]$

5 Set $\texttt{last\_update} = T + 1$

6 **If** $\texttt{Cond}(\mathcal{BC}) = 0$, then

7 **Return** "Error: Condition not met"

Protocol 8: $T+1$-OTP' Instantiation

Setup$(\mathcal{BC}_\ell, \texttt{Cond})$:

Post smart contract $\mathcal{SC}$ as defined in Smart-Contract 5 to the blockchain at time ℓ

Let $T_{\textbf{Init}} = \ell$.

Let $\mathrm{id}_{\textbf{Setup}} = (\texttt{idx}_1, ..., \texttt{idx}_r, \texttt{addr})$ be the identifier of the smart contract and $\texttt{addr}$ be the address of the smart contract. Also, let ℓ be the block time of the smart contract's initialization.

Return $\mathrm{id}_{\textbf{Setup}}, T_{\textbf{Init}}$

Init$\big(V_{T+1}, C, \mathrm{id}_{\textbf{Setup}} = (\texttt{idx}_1, ..., \texttt{idx}_r, \texttt{addr}), T_{\textbf{Init}}\big)$ for $C : \{0,1\}^v \times \{\bot\} \cup \{0,1\}^r \to \mathcal{Y}$ where $\mathcal{V} \subseteq \{0,1\}^v$

Define $C' : \{0,1\}^v \times \{0,1\}^r \times \{0,1\} \to \mathcal{Y}$ such that $C'(V, x, 0) = C(V, \bot)$ and $C'(V, x, 1) = C(V, x)$

$G, \left(\left\{\tilde{w}_j^0, \tilde{w}_j^1\right\}_{\{j\in[v]\}}, \{w_i^0, w_i^1\}_{\{i\in[r]\}}, \{F_b\}_{\{b\in\{0,1\}\}}\right) \leftarrow \texttt{GC.Garble}(C')$[10]

For $i \in [r], b \in \{0,1\}$

Let $\mathrm{id}_i^b \leftarrow \texttt{SecretStore}\big(w_i^b, V_{T+1}, \texttt{idx}_i, b\big)$

For $j \in [v], b \in \{0,1\}$

Let $\tilde{\mathrm{id}}_j^b \leftarrow \texttt{SecretStore}\big(\tilde{w}_j^b, V_{T+1}, \texttt{idx}_{\mathrm{ver}_j}, b\big)$ where $\texttt{idx}_{\mathrm{ver}_j}$ is the position of the j-th bit for the string of next verification keys.

Let $\mathrm{id}_{F_0} \leftarrow \texttt{SecretStore}(F_0, V_{T+1}, \texttt{last\_update}, T_{\textbf{Init}})$

Let $\mathrm{id}_{F_1} \leftarrow \texttt{SecretStore}(F_1, V_{T+1}, \texttt{last\_update}, T+1)$

Return $\mathrm{id}_{\textbf{Init}} = \left(G, \left\{\tilde{\mathrm{id}}_j^b\right\}_{\{j\in[v],b\in\{0,1\}\}}, \left\{\mathrm{id}_i^b\right\}_{i\in[r],b\in\{0,1\}}, \left\{\mathrm{id}_{F_0}, \mathrm{id}_{F_1}\right\}\right)$

SetInput$\big(\mathcal{BC}_T, \mathrm{id}_{\textbf{Setup}}, x\big)$:

Call **SetInput** on $\mathcal{SC}$ at address $\texttt{addr}$ with the input x

If $\mathcal{SC}$ returns an error

Return the same error

Eval$\Big(\mathcal{BC}_{T+1}, \mathrm{id}_{\textbf{Init}} = \Big(G, \big\{\mathrm{id}_i^b\big\}_{i\in[r],b\in\{0,1\}}\Big)\Big)$:

For $i \in [r]$

Let $w_i^{x_i} \leftarrow \texttt{SecretRelease}\big(\mathrm{id}_i^{x_i}\big)$

If $\texttt{last\_update} = T+1$:

Let $F \leftarrow \texttt{SecretRelease}\big(\mathrm{id}_{F_1}\big)$

Else

Let $F \leftarrow \texttt{SecretRelease}\big(\mathrm{id}_{F_0}\big)$

For $j \in [v]$ and string representing the next verification keys $V \in \{0,1\}^v$,

Let $\tilde{w}_j^{V_j} \leftarrow \texttt{SecretRelease}\Big(\tilde{\mathrm{id}}_j^{V_j}\Big)$

Call $y = \texttt{GC.Eval}\left(G, \left(\left\{\tilde{w}_j^{V_j}\right\}_{j\in[v]}, \{w_i^{x_i}\}_{i\in[r]}, F\right)\right)$

Return y

[10] F_b stands for the wires on the "flag" bit which represents if the input is $\bot$ or not.

Lemma 5.1 *(Soundness of modified $T+1$-OTP, Protocol 8)*: The protocol above is a sound $T+1$-conditional one-time program, $T+1$-OTP$'$ assuming the soundness of the underlying blockchain, SWE, and WE for KZG commitments, where soundness is defined as in Definition 5.4.

Proof sketch: The proof follows in almost the same way as Theorem 5.1 except for one key difference: if the input is not set, we can decrypt the wires corresponding to the input $\perp$. First, we can see that, per-bit of input, only one of the wire labels can be decrypted by the security of the blockchain. Next, if F_0 is released, then the smart contract was not updated, and the user will be able to evaluate C on $\perp$. If F_1 is released, then the smart contract was updated and the user will be able to evaluate C on x. ■

6 Conditional RAM Obfuscation

Stateful obfuscation is a powerful primitive that allows for the obfuscation of a program such that it can be executed on a RAM machine where the memory itself is "authenticated" in such a way that the program can not be "reversed" [3].

We recall the simulation-based definition of RAM obfuscation from [3].

Definition 6.1 *(RAM-Oracle)*: A stateful RAM-oracle for a program P, connoted R$\mathcal{O}$, with authenticated RAM, is an oracle which maintains an updatable state across queries. Specifically, we model any algorithm $\mathcal{A}$ with oracle access to R$\mathcal{O}_p$ and starting state RAM_0 as follows: the algorithm's queries to the oracles, $x_1, ..., x_\ell$ with outputs $y_1, ..., y_\ell$, can be broken down as follows

$$(y_i, \text{RAM}_i) = P(x_i, \text{RAM}_{i-1}). \tag{21}$$

where RAM_i is the updated state of the oracle after query x_i.

As with our one-time program, we will change the definition of RAM obfuscation to allow for conditional execution. Moreover, we will change the API from [3] to better fit the blockchain context.

We then say that a scheme is a RAM-blackbox obfuscator if it can be used to construct a RAM-oracle for any program P. Specifically, we use the following definition:

Definition 6.2 *(RAM-Blackbox Obfuscator)*: A RAM-blackbox obfuscator is a tuple of algorithms (R$\mathcal{O}$.`send`, R$\mathcal{O}$.`eval`) such that:

- **Setup**$(\mathcal{BC}_\ell, \texttt{Cond}, r)$ for $\ell \leq T$ is a setup algorithm that takes as input a blockchain at time ℓ, circuit input size r, and a condition `Cond` and outputs an identifier, $\text{id}_{\textbf{Setup}}$.
- **Init**$\big(\mathcal{BC}_T, P, \text{id}_{\textbf{Setup}}, \text{RAM}_0\big)$ is an initialization algorithm that outputs an initial identifier, id_0 with initial state RAM_0
- **SetInput**$\big(\mathcal{BC}_T, \text{id}_{\textbf{Setup}}, x\big)$ is an algorithm that takes as input a value x and sets the input of the OTP to x by updating the blockchain at time T.
- **Eval**$\big(\mathcal{BC}_{T+p+1}, \text{id}_p\big)$ is an algorithm that takes as input the blockchain at time $T+1$ and attempts to output $P\big(x, \text{RAM}_p\big)$. **Eval** also returns id_{p+1} which is the identifier of the next RAM-oracle.

Then, we define the security of a RAM-blackbox obfuscator as follows:

Definition 6.3 *(RAM-Blackbox Obfuscator Simulation Soundness)*:

A RAM-blackbox obfuscator is secure if for every polynomial-time adversary $\mathcal{A}$ there exists a polynomial-time simulator Sim such that:

$$\begin{aligned}\mathcal{A}\big(1^\lambda, 1^{|P|}, \mathcal{BC}_{T+p+1}, \mathrm{id}_{\mathbf{Setup}} \leftarrow \mathbf{Init}(\mathcal{BC}_\ell, \mathtt{Cond}, r), \mathtt{Cond}, \mathbf{Setup}\big(\mathcal{BC}_T, P, \mathrm{id}_{\mathbf{Setup}}\big)\big)\\ \overset{c}{\approx} \mathsf{Sim}^{\mathrm{R}\mathcal{O}(P)}\big(1^\lambda, 1^{|P|}, \mathtt{Cond}, \mathbf{Setup}(\mathcal{BC}_\ell, \mathtt{Cond}), \mathcal{BC}_{T+p+1}\big)\end{aligned} \tag{22}$$

where $\mathrm{R}\mathcal{O}(P)$ is the RAM-oracle for program P.

6.1 Instantiating RAM Obfuscation

We now show how we can use Definition 5.3 to construct a RAM obfuscator.

First, we need to define a "recursive" circuit for RAM obfuscation, in a similar way to Ref. [3]. We note that our RAM programs have some notion of time (denoted as subscript i) which is the last block in which the RAM program was executed. Upon execution of the RAM program, the time is increased to $T+1$.

Definition 6.4 *(Recursive RAM Encoding)*: For circuit class $\mathcal{P} : \{0,1\}^r \times \{0,1\}^m \to \mathcal{Y} \times \{0,1\}^m$ where $\{0,1\}^m$ represents the "internal" state of the RAM, we define a recursive circuit $\overline{P'}_i$ for $P'_i = P(\cdot, \mathrm{RAM}_i)$ and $P \in \mathcal{P}$ as follows: Let $(y, \mathrm{RAM}_{T+1}) = P'_{T(x)}$ where $x \in \{0,1\}^r$ and RAM_{T+1} is the next internal state of the RAM. Moreover, let $\mathtt{vk}_1, ..., \mathtt{vk}_n$ be the verification keys for $T+1$-th block. Then, we define the recursive circuit as follows for $x \in \{\bot\} \cup \{0,1\}^r$: if $x \neq \bot$, then for verification keys V_{T+2} for the $T+2$-th block:

$$\begin{aligned}&\overline{P'_T}(V_T, x) = \\ &\qquad \big(P'(x), T+1\text{-}\mathsf{OTP}'.\mathbf{Init}\big(V_{T+1}, \overline{P'}_{T+1}, \mathrm{id}_{\mathbf{Setup}}, T\big)\big) = \big(y, \mathrm{id}^{T+1}_{\mathbf{Init}}\big).\end{aligned} \tag{23}$$

Otherwise, if $x = \bot$, then we define the circuit as:

$$\overline{P'_T}(V_T, \bot) = \big(\bot, T+1\text{-}\mathsf{OTP}'.\mathbf{Init}\big(V_{T+1}, \overline{P'_{T+1}}, \mathrm{id}_{\mathbf{Setup}}, T_{\mathrm{Last}}\big)\big) = \big(\bot, \mathrm{id}^{T+1}_{\mathbf{Init}}\big) \tag{24}$$

where T_{Last} is the prior block time, T, for which the RAM program was executed on an input which was not $\bot$.

Finally, now that we have setup our technical machinery, we can define our RAM obfuscation protocol in a relatively straightforward manner.

Protocol 9: RAM Obfuscation for circuit class $\mathcal{P} : \{0,1\}^r \times \{0,1\}^m \to \mathcal{Y} \times \{0,1\}^m$ where $\{0,1\}^m$ represents the "internal" state of the RAM

Setup$(\mathcal{BC}_\ell, \texttt{Cond})$:

Call $\mathrm{id}_{\mathbf{Setup}}, T_{\mathbf{Init}} \leftarrow T + 1\text{-}\mathsf{OTP}'.\mathbf{GlobalSetup}\ (\mathcal{BC}_\ell, \texttt{Cond}, r)$

Init$\big(\mathcal{BC}_T, P, \mathrm{id}_{\mathbf{Setup}} = (\texttt{idx}_1, ..., \texttt{idx}_r, \texttt{addr}), \mathrm{RAM}_0, T_{\mathbf{Init}}\big)$:

Let $P' = P(\cdot, \mathrm{RAM}_0)$ and

$$\mathrm{id}^T_{\mathbf{Init}} = T + 1\text{-}\mathsf{OTP}'.\mathbf{Init}\big(V_{T+1}, \overline{P'_T}, \mathrm{id}_{\mathbf{Setup}}, T_{\mathbf{Init}}\big) \tag{25}$$

where $\overline{P'}$ is the recursive RAM encoding of P' for time $T+1$ (Definition 6.4).

Return $\mathrm{id}^T_{\mathbf{Init}}$

SetInput$\big(\mathcal{BC}_T, \mathrm{id}_{\mathbf{Setup}}, x\big)$:

Call $T + 1\text{-}\mathsf{OTP}'.\mathbf{SetInput}\big(\mathcal{BC}_T, \mathrm{id}_{\mathbf{Setup}}, x\big)$

If the smart contract, $\mathcal{SC}$, returns an error

Return the same error

Eval$\big(\mathcal{BC}_{T+1}, \mathrm{id}^T_{\mathbf{Init}}\big)$:

Call $y, \mathrm{id}^{T+1}_{\mathbf{Init}} \leftarrow T + 1\text{-}\mathsf{OTP}'.\mathbf{Eval}\big(\mathcal{BC}_{T+1}, \mathrm{id}^T_{\mathbf{Init}}\big)$

Return y and publish $\mathrm{id}^{T+1}_{\mathbf{Init}}$

Theorem 6.1 (RAM-Blackbox Obfuscator): The RAM-blackbox obfuscator $(\mathrm{R}\mathcal{O}.\texttt{send}, \mathrm{R}\mathcal{O}.\texttt{eval})$ is a secure RAM obfuscator as defined in Definition 6.4 assuming the soundness of the one-time program $T + 1\text{-}\mathsf{OTP}'$ and underlying blockchain.

We provide a proof of the above theorem in Appendix C and a sketch of the proof below.

Proof sketch: The proof follows along the lines of Ref. [3]. As we have a one-time program from the previous section, we can use the one-time programs simulation soundness to show that only one query per RAM state is possible. Then, we recursively replace each call to one-time program with its simulation based definition. So, for a polynomial number of queries to the RAM oracle, we can simulate each query using the one-time program. ■

7 Conclusion

In this work, we show how to construct some of the most powerful cryptographic primitives where the security is based on the underlying security of a blockchain. Specifically, we construct a $T + 1$-eWEB scheme, a conditional one-time program scheme, and a conditional RAM obfuscation scheme. Along the way, we give a simple yet formal description of a proof of stake blockchain model as well.

Though our construction is relatively simple, it is not without its drawbacks; follow up work is needed to improve the construction. First, our construction requires a modified BLS signature scheme with $\lambda_{\mathbb{G}}$ multiplicative blowup in the signature size. We are thus left with the following open questions:

Can we construct a scheme without $\lambda_{\mathbb{G}}$ multiplicative blowup in the signature size?

It seems as though we can use the aggregation of messages in BLS to reduce the size of the signature though this would require modifying the underlying SWE scheme.

Moreover, in order to instantiate our RAM obfuscation scheme, we require a **bit-by-bit** representation of the verification keys in the blockchain state, which can lead to a large size blowup in the blockchain state. So, we have the next open question:

Can we construct a scheme that does not require a bit-by-bit representation of the verification keys?

Moreover, our construction is heavily based on pairing-based cryptography, a notably **not post-quantum secure primitive**. Unlike a blockchain, which uses cryptography for consensus, our primitives can also be used for privacy. Thus, our work is vulnerable to "store-now-decrypt-later" attacks, where an adversary can store ciphertexts and decrypt them later once a functional quantum computer is available. So, we are left with the following open questions:

Can we construct a scheme that is post-quantum secure while simultaneously keeping its simplicity?

Finally, we note that our construction is relatively inefficient. For example, on the Ethereum blockchain, the set of signers does not *completely* change from block to block. Thus, we could potentially improve the efficiency of our scheme by having some update rule for the set of signers rather than having to feed the whole set of signers into our garbled circuits.

Acknowledgements

The author is grateful to the helpful discussions and feedback from Janmajaya Mall and Justin Drake. The author also acknowledges funding and support from Neon Tetra LLC and from the NSF Graduate Research Fellowship Program (NSF-GRFP).

Bibliography

[1] S. Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, (n.d.).

[2] R. Goyal and V. Goyal, Overcoming Cryptographic Impossibility Results Using Blockchains, (2017).

[3] L. Stambler, Cryptography without Long-Term Quantum Memory and Global Entanglement, (2025).

[4] V. Goyal, A. Kothapalli, E. Masserova, B. Parno, and Y. Song, Storing and Retrieving Secrets on a Blockchain, (2020).

[5] F. Benhamouda, C. Gentry, S. Gorbunov, S. Halevi, H. Krawczyk, C. Lin, T. Rabin, and L. Reyzin, Can a Public Blockchain Keep a Secret?, (2020).

[6] C. Gentry, S. Halevi, and V. Lyubashevsky, *Practical Non-Interactive Publicly Verifiable Secret Sharing with Thousands of Parties*, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2022), pp. 458–487.

[7] C. Gentry, S. Halevi, H. Krawczyk, B. Magri, J. B. Nielsen, T. Rabin, and S. Yakoubov, *Yoso: You Only Speak Once: Secure Mpc with Stateless Ephemeral Roles*, in *Annual International Cryptology Conference* (2021), pp. 64–93.

[8] V. Goyal, A. Kothapalli, E. Masserova, B. Parno, and Y. Song, *Storing and Retrieving Secrets on a Blockchain*, in *IACR International Conference on Public-Key Cryptography* (2022), pp. 252–282.

[9] A. Kate, G. M. Zaverucha, and I. Goldberg, Constant-Size Commitments to Polynomials and Their Applications, Advances in Cryptology - ASIACRYPT 2010 **6477**, 177 (2010).

[10] D. Boneh, B. Lynn, and H. Shacham, Short Signatures from the Weil Pairing, (n.d.).

[11] N. Döttling, L. Hanzlik, B. Magri, and S. Wohnig, McFly: Verifiable Encryption to the Future Made Practical, (2022).

[12] N. Fleischhacker, M. Hall-Andersen, and M. Simkin, Extractable Witness Encryption for KZG Commitments and Efficient Laconic OT, (2024).

[13] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, One-Time Programs, Advances in Cryptology – CRYPTO 2008 **5157**, 39 (2008).

[14] M. Bellare, V. T. Hoang, and P. Rogaway, *Foundations of Garbled Circuits*, in *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (ACM, Raleigh North Carolina USA, 2012), pp. 784–796.

[15] A. C.-C. Yao, *How to Generate and Exchange Secrets*, in *27th Annual Symposium on Foundations of Computer Science (Sfcs 1986)* (1986), pp. 162–167.

A Proofs for T+1-eWEB

We now provide the missing proof for Section 4 and Theorem 4.2. We restate the theorem for convenience.

Theorem 1.1 (Soundness of Protocol 2): Protocol 2 is sound (Definition 4.2) assuming that the signature-based witness encryption is sound, and the KZG base witness encryption is sound.

Proof: The proof proceeds via contrapositive. Assume that there is no PPT extractor, E, which can produce two distinct signatures for the same bit of the KZG commitment for all $j \in [\lambda_{\mathbb{G}}]$ and that E cannot produce $\texttt{state}^*$ which is a valid blockchain state and meets the $T+1$-eWEB condition.

We now proceed to show that the adversary cannot distinguish between the two messages s and s' for any fixed $\texttt{idx}$ and $\texttt{target}$ given blockchain $\mathcal{BC}$ at block $F \geq T$ via a series of hybrid arguments.

- Hyb_0: The first hybrid is the real protocol where the adversary receives the output of $\texttt{SecretStore}$ and the blockchain $\mathcal{BC}_F$ for $F \geq T$
- Hyb_1: The second hybrid is the same except that if $F = T$, we replace all the wire labels in the encryption with random values, i.e. $\hat{w}_j^0, \hat{w}_j^1$ are replaced with random values for $j \in [\lambda_{\mathbb{G}}]$.
- Hyb_2: The same as Hyb_1 except that if $F > T$, then let σ be the signature for the $T+1$ block. Then let $\overline{\sigma}$ be the negation of σ ($\overline{\sigma_j} = 1 - \sigma_j$). Then, set $\hat{w}_j^{\overline{\sigma}_j} = \hat{r}_j$ where $\hat{r}_j$ is a random value for $j \in [\lambda_{\mathbb{G}}]$.
- Hyb_3: The same as Hyb_2 except that we replace the garbled circuit $\hat{G}$ with its simulator:
  - Let $\hat{G}$ be the garbled circuit for the first stage circuit Circuit 4 and the adversary's view is
    $$\hat{G}, \left\{\hat{w}_j^{x_j}\right\}_{j \in [\lambda_{\mathbb{G}}]}, \texttt{ct}, \mathcal{BC}, \texttt{idx}, \texttt{target} \tag{26}$$
    and gets replaced with simulator Sim_3 with view
    $$1^\lambda, 1^n, 1^{|C|}, C(x), \mathcal{BC}, \texttt{idx}, \texttt{target} \tag{27}$$
    where Sim_3 first simulates $\texttt{ct}$ and then uses the garbled circuit simulator. If $F = T$, then we set $C(x) = \bot$.
- Hyb_4: The same as Hyb_3 except that if $F > T$, we replace $\widehat{\texttt{ct}}, \texttt{ct} = C(x)$ with $\widehat{\texttt{ct}}, \texttt{ct}'$ where $\texttt{ct}'$ is an encryption of a random value.

Then, we can see in Hyb_4, as either $C(x) = \bot$ or $C(x)$ output $\texttt{ct}'$ which is an encryption of a random value, the adversary cannot distinguish between s and s' as we have removed s from the view of the adversary. ■

Lemma 1.1 *($\mathsf{Hyb}_0 \overset{c}{\approx} \mathsf{Hyb}_1$)*: The first hybrid, Hyb_0, is computationally indistinguishable from the second hybrid, Hyb_1, we have a blockchain at time T and thus no signature is revealed for the $T+1$ block. And so, by the security of the signature-based witness encryption (Definition 2.5), the wire labels $\hat{w}_j^0, \hat{w}_j^1$ are indistinguishable from encryptions of random messages for all $j \in [\lambda_{\mathbb{G}}]$.

Lemma 1.2 *(*$\mathsf{Hyb}_1 \overset{c}{\approx} \mathsf{Hyb}_2$*)*: Note that by our assumption, the extractor E cannot produce two distinct signatures for the same bit of the KZG commitment for all $j \in [\lambda_{\mathbb{G}}]$. And so, by the security of the signature-based witness encryption (Definition 2.5), *one* of the wire labels $\hat{w}_j^0, \hat{w}_j^1$ are indistinguishable from encryptions of random messages for all $j \in [\lambda_{\mathbb{G}}]$. Because the signature σ is revealed, then the indistinguishable wire labels correspond to $\overline{\sigma}$

Lemma 1.3 *(*$\mathsf{Hyb}_2 \overset{c}{\approx} \mathsf{Hyb}_3$*)*: Define the circuit family for our garbled circuit $\mathcal{C}$ as

$$\mathcal{C} = \left\{ C(\texttt{comm}) = \texttt{EncrWithKZG}_{\texttt{idx, target},s}(\texttt{comm}) \mid s \in \mathcal{S} \right\} \tag{28}$$

where the subscript indicates the fixed values of the circuit. Then, by the previous two hybrids, at least one of the wire labels $\hat{w}_j^{\overline{\sigma}_j}$ is indistinguishable from random for all $j \in [\lambda_{\mathbb{G}}]$. And so, the simulator can choose a random value for $\hat{w}_j^{\overline{\sigma}_j}$ and encrypt it using the blockchain $\mathcal{BC}$. As the blockchain itself is independent of $\mathcal{C}$ as well, then we can use the selective security of the garbled circuits (Definition 2.13) to replace the garbled circuit $\hat{G}$ with its simulator.

Lemma 1.4 *(*$\mathsf{Hyb}_3 \overset{c}{\approx} \mathsf{Hyb}_4$*)*: If $F = T$, then $C(x) = \bot$, and so s is removed from the view of the adversary. If $F > T$, then we have $\widehat{\texttt{ct}}, \texttt{ct} \leftarrow C(\texttt{comm})$. Note that $\widehat{\texttt{ct}}$ can only be decrypted to reveal k if an extractor can produce a valid proof π for $\texttt{state}_{T+1}[\texttt{idx}] = \texttt{target}$. Moreover, note that we can only run $C(\texttt{comm})$ once on a valid signature for block $T+1$. And so, $\texttt{comm}$ must pass verification for $\texttt{state}_{T+1}$ (i.e. $\text{Verify}_{V(\texttt{state}_{T+1})} = 1$) and, by our assumption, $\texttt{state}_{T+1}[\texttt{idx}] \neq \texttt{target}$. Assume towards contradiction that the adversary can decrypt k and so an extractor exists, E', which can produce a valid proof π for $\texttt{state}_{T+1}[\texttt{idx}] = \texttt{target}$ in polynomial time. And so, either the adversary has state $\texttt{state}'$ which has commitment $\texttt{comm}$ and $\texttt{state}' \neq \texttt{state}^*$, or the adversary has $\texttt{state}^*$ which has commitment $\texttt{comm}$ and $\texttt{state}^*[\texttt{idx}] \neq \texttt{target}$. In the first case, the adversary breaks the binding property of the KZG commitment as it can produce two distinct signatures for the same bit of the KZG commitment. In the second case, the adversary can produce a valid proof π for $\texttt{state}_{T+1}[\texttt{idx}] = \texttt{target}$ in polynomial time, breaking the soundness of the KZG commitment.

B Proofs for T+1-OTP

We now provide the missing details for the proof of Theorem 5.1. Recall the theorem statement:

Theorem 2.1 (One-Time Program Soundness from $T+1$-eWEB): Assuming the soundness of our $T+1$-eWEBs and that only one blockchain state can be produced per block, the protocol above is a sound $T+1$-conditional one-time program, $T+1$-OTP, where soundness is defined as in Definition 5.2.

We note that the following proof is a rather straightforward application of the soundness of the $T+1$-eWEB (Definition 4.2) and the security of the garbled circuits (Definition 2.13).

Proof: We proceed via a series of hybrid arguments.

- Hyb_0: the real protocol, here the view of the adversary is

$$\left\{ 1^\lambda, 1^{|C|}, \mathcal{BC}_{T+1}, \text{id}_{\textbf{Setup}} \leftarrow \textbf{Setup}(\mathcal{BC}_\ell, \texttt{Cond}), \texttt{Cond}, \textbf{Init}\left(V_{T+1}, C, \text{id}_{\textbf{Setup}}\right) \right\} \tag{29}$$

- Hyb_1: For all i, $\mathcal{BC}_{T+1}[\mathtt{idx}_i] = -1$, replace both w_i^0 and w_i^1 with random values.
- Hyb_2: If for all i, $\mathcal{BC}_{T+1}[\mathtt{idx}_i] \neq -1$, let $x_i = \mathcal{BC}_{T+1}[\mathtt{idx}_i]$. Replace $w_i^{1-x_i}$ with a random value.
- Hyb_3: If $\mathcal{BC}_{T+1}[\mathtt{idx}_i] \neq -1$ for all i, replace the garbled circuit with its simulator, giving away $C(x)$. We now have the simulated view

$$\left\{1^\lambda, 1^{|C|}, C(x), \mathtt{Cond}, \mathbf{Setup}(\mathcal{BC}_\ell, \mathtt{Cond}), \mathcal{BC}_{T+1}\right\}. \tag{30}$$

We can then see that in Hyb_4, if $\mathtt{Cond}(\mathtt{state}_{T+1}) = 0$, then, by the soundness of the blockchain, $\mathcal{BC}_{T+1}[\mathtt{idx}_i] = -1$ for all $i \in [r]$ and so the garbled circuit is indistinguishable from random by Definition 2.14. Otherwise, if $\mathtt{Cond}(\mathtt{state}_{T+1}) = 1$, by the security of the eWEB (Definition 4.2) and Definition 2.13, the simulator can only get one evaluation of C under input x. ■

Lemma 2.1: $\mathsf{Hyb}_0 \overset{c}{\approx} \mathsf{Hyb}_1$

Proof: The first hybrid follows from the soundness of the eWEB (Definition 4.2) as we can replace the message with a random value if the blockchain's state at time $T+1$ does not match the target. ■

Lemma 2.2: $\mathsf{Hyb}_1 \overset{c}{\approx} \mathsf{Hyb}_2$

Proof: The second hybrid follows from the soundness of the eWEB (Definition 4.2) as well: if $\mathcal{BC}_{T+1}[\mathtt{idx}_i] = x_i$ can only equal 0 or 1 and thus by the soundness of the eWEB, $w_i^{1-x_i}$ is removed from the view of the adversary as we can replace its encryption with an encryption of a random value. ■

Lemma 2.3: $\mathsf{Hyb}_2 \overset{c}{\approx} \mathsf{Hyb}_3$

Proof: Note that the adversary's view can only contain at most one set of wires, $\{w_i^{x_i}\}_{\{i\in[r]\}}$. So, we can replace the garbled circuit with its simulator, which outputs $C(x)$, by the soundness of the garbled circuit (Definition 2.13). ■

C Proof for RAM-Orcale Construction

Here, we will provide a proof of soundness for our RAM obfuscation scheme. As before, we restate the theorem for convenience:

Theorem 3.1 (RAM-Blackbox Obfuscator): The RAM-blackbox obfuscator $(\mathrm{R}\mathcal{O}.\mathtt{send}, \mathrm{R}\mathcal{O}.\mathtt{eval})$ is a secure RAM obfuscator as defined in Definition 6.4 assuming the soundness of the one-time program $T+1$-OTP' and underlying blockchain.

The proof will follow the proof of RAM obfuscation in Ref. [3] closely. The high level idea will be to use a series of induction steps, alongside the strong notion of one-time simulation security for our $T+1$-OTP' to replace each call to the one-time program with a call to two separate programs: one which outputs the program's result and the other which outputs the setup for the next one-time program. We then recursively simulate the next-one time program.

Proof: We proceed via a series of inductive hybrids. Specifically, for $\ell \in \mathrm{poly}(\lambda)$, we will show that

$$\mathcal{A}\big(1^\lambda, 1^{|P|}, \mathcal{BC}_{T+p+1}, \mathrm{id}_{\mathbf{Setup}} \leftarrow \mathbf{Init}(\mathcal{BC}_\ell, \mathtt{Cond}, r), \mathtt{Cond}, \mathbf{Setup}\big(\mathcal{BC}_T, P, \mathrm{id}_{\mathbf{Setup}}\big)\big) \overset{c}{\approx}$$
$$S_\ell \circ (\mathbb{I}, \mathtt{Single}(P(\mathrm{RAM}_{\ell-1}, \cdot)) \circ \tag{31}$$
$$... \circ (\mathbb{I}, \mathtt{Single}(P(\mathrm{RAM}_1, \cdot)) \circ S_1 \circ (\mathbb{I}, \mathtt{Single}(P(\mathrm{RAM}_0, \cdot))) \circ S_0.$$

for PPT algorithms S where $\ell \in \mathrm{poly}(\lambda)$.

Consider the following sequence of hybrids:

- Hyb_0: the real protocol
- Hyb_1: replace $\mathcal{A}$ with the simulator for the $T+1$-OTP': $S^{\mathtt{Single}\left(\overline{P'_{T+1}}\right)}$. Then break up S into 2, $S_0, \hat{S}_1$ such that
$$S = \hat{S}_1^{\mathtt{Single}\left(T+1-\mathsf{OTP}'.\mathbf{Init}\left(\cdot, \overline{P'_1}, \mathrm{id}_{\mathbf{Setup}}, T_{\mathrm{Last}}\right)\right)} \circ S_0^{\mathtt{Single}(P(\mathrm{RAM}_0, \cdot))}. \tag{32}$$
In words, we split the simulator into two parts, one which returns the output of the program $P(\mathrm{RAM}_0, \cdot)$ and the other which returns the setup for the next one-time program, $T+1$-OTP'.
- Hyb_i for $i \in \{2, ..., \ell'\}$ for some $\ell' \geq \ell$ choosen later in the proof. Replace the simulator, $\hat{S}_{i-1}$ with $S^{\mathtt{Single}\left(\overline{P'}_{\mathrm{RAM}_{i-1}}\right)}$. Then, take $S^{\mathtt{Single}\left(\overline{P'}_{\mathrm{RAM}_{i-1}}\right)}$ and replace it with the two simulators, $S_{i-1}, \hat{S}_i$ such that
$$S^{\mathtt{Single}\left(\overline{P'_{i-1}}\right)} = \hat{S}_i^{\mathtt{Single}\left(T+1-\mathsf{OTP}'.\mathbf{Init}\left(\cdot, \overline{P'_i}, \mathrm{id}_{\mathbf{Setup}}, T_{\mathrm{Last}}\right)\right)} \circ S_{i-1}^{(\mathtt{Single}(P(\mathrm{RAM}_{i-1}, \cdot)).} \tag{33}$$

To see that the above hybrids are valid, note that $\overline{P'_i}$ simultaneously evaluates P'_i, outputs $\mathrm{id}_{\mathbf{Init}}^{i+1}$ via a call to $T+1-\mathsf{OTP}'.\mathbf{Init}$. So, a simulator which can first evaluate $P(\mathrm{RAM}_i, \cdot)$ and then evaluate $T+1-\mathsf{OTP}'.\mathbf{Init}\left(\cdot, \overline{P'_i}, \mathrm{id}_{\mathbf{Setup}}, T_{\mathrm{Last}}\right)$ is strictly stronger than (and can thus simulate) the adversary in the real protocol.

Finally, note that after the ℓ'-th hybrid, we have
$$S = \hat{S}_{\ell'}^{\mathtt{Single}\left(T+1-\mathsf{OTP}'.\mathbf{Init}\left(\cdot, \overline{P'_{\ell'}}, \mathrm{id}_{\mathbf{Setup}}, T_{\mathrm{Last}}\right)\right)} \circ S_{\ell'-1}^{\mathtt{Single}(P(\mathrm{RAM}_{\ell'-1}, \cdot))} \circ \\ ... \circ S_1^{\mathtt{Single}(P(\mathrm{RAM}_1, \cdot))} \circ S_0^{\mathtt{Single}(P(\mathrm{RAM}_0, \cdot))} \tag{34}$$

Given that S runs in polynomial time, we can always find some polynomially large $\ell' \geq \ell$ such that $\hat{S}_{\ell'}$ is the null simulator as S's runtime is consumed by $S_{\ell-1}, ..., S_0$.

We thus have our desired result as we can re-write S as a set of RAM oracle calls:
$$S_\ell \circ (\mathbb{I}, \mathrm{R}\mathcal{O}[P, \mathrm{RAM}_{\ell-1}]) \circ ... \circ (\mathbb{I}, \mathrm{R}\mathcal{O}[P, \mathrm{RAM}_1]) \circ S_1 \circ (\mathbb{I}, \mathrm{R}\mathcal{O}[P, \mathrm{RAM}_0]) \circ S_0. \tag{35}$$

■