BABE: Verifying Proofs on Bitcoin Made 1000x Cheaper

Sanjam Garg[1,2], Dimitris Kolonelos[1], Mikhail Sergeevitch[3], Srivatsan Sridhar[4], and David Tse[4,5]

[1]University of California, Berkeley
[2]Exponential Science Foundation
[3]Babylon Labs
[4]Byzantine Research
[5]Stanford University

Abstract

Endowing Bitcoin with the ability to verify succinct proofs has been a longstanding problem with important applications such as scaling Bitcoin and allowing the Bitcoin asset to be used in other blockchains trustlessly. It is a challenging problem due to the lack of expressiveness in the Bitcoin scripting language and the small Bitcoin block space. BitVM2 [LAA+25] is the state-of-the-art verification protocol for Bitcoin used in several mainnets and testnets [Bit25a, Cit25, BOB25a], but it suffers from very high on-chain Bitcoin transaction fees in the unhappy path (over $\$14,000$ in a recent experiment [LAA+25]). Recent research BitVM3 dramatically reduces this on-chain cost by using a garbled SNARK verifier circuit to shift most of the verification off-chain [Rub24, Lin24], but each garbled circuit is 42 GiBytes in size, so the off-chain storage and setup costs are huge. This paper introduces BABE, a new proof verification protocol on Bitcoin, which preserves BitVM3's savings of on-chain costs but reduces its off-chain storage and setup costs by three orders-of-magnitude. BABE uses a witness encryption scheme for linear pairing relations [GKPW24] to verify Groth16 proofs. Since Groth16 verification involves non-linear pairings, this witness encryption scheme is augmented with a secure two-party computation protocol implemented using a very efficient garbled circuit for scalar multiplication on elliptic curves. The design of this garbled circuit builds on the recent work of Argo MAC [EL26], a garbling primitive that efficiently computes homomorphic MACs on such curves.

Contents

1 Introduction

1.1 Motivating Example

Bob has 1 BTC and would like to use it as collateral to borrow \$50K worth of stablecoins for real-world use. Currently Bob's only options are to go to a centralized entity such as Tether to directly borrow, or to a centralized entity like Coinbase which mints Bob a wrapped asset cbBTC that he can use in a smart contract lending protocol on Ethereum or other chains. However, in both cases the centralized entity would take custody of Bob's BTC, violating the core ethos of Bitcoin: *trustlessness.*[1] If Bob had 1 ETH, on the other hand, he would be able to participate directly in a trustless lending protocol like Aave on Ethereum without going through a centralized entity. Unfortunately, the Bitcoin chain does not support smart contract protocols such as Aave. This begs the question:
How do we allow the \$1.8 *trillion worth of BTC to participate trustlessly in smart contracts like Aave on chains like Ethereum?*

Consider what such a protocol would look like:

1. Borrower Bob deposits his BTC using a Bitcoin transaction;
2. A lending position is created on Ethereum with the BTC as collateral and \$50K in stablecoin is lent from a lender Larry to Bob;
3. If Bob returns the loan, then the BTC collateral can be withdrawn by Bob;
4. If Larry liquidates the loan because BTC price has dropped below a threshold, then the BTC collateral can be withdrawn by Larry.

To achieve step 2 trustlessly, Bob has to prove a certain state of the Bitcoin chain to Ethereum. This proof can be verified by smart contracts which already exist on Ethereum [Cat25]. To achieve steps 3 and 4 trustlessly, each party has to prove a certain state of the Ethereum chain to Bitcoin so that he can withdraw. Due to the rudimentary nature of the Bitcoin scripting language, however, even a succinct proof verifier takes about 900 Mbytes of Bitcoin block space, equivalent to 225 Bitcoin blocks [LAA$^+$25]. Moreover, this verifier has to be submitted on-chain as a Bitcoin transaction every time a proof needs to be verified, thus rendering it totally impractical (the transaction fees will be millions of dollars).

Note that the lending example is only one application. The ability to verify proofs is important for any application in which the withdrawal of the deposited BTC depends on the state of another chain, such as light-client based Bitcoin bridges to rollups and other chains [LAA$^+$25].

1.2 Verifying Proofs on Bitcoin

1.2.1 BitVM and BitVM2

BitVM [Lin23, AAL$^+$24] initiated a line of work to reduce the on-chain footprint of proof verification using optimistic methods. It is a two-party protocol between the Prover and the Verifier, with the Verifier helping Bitcoin verify the Prover's proof by challenging the Prover if its proof is invalid. This protocol ends if either the Prover wins, i.e. convinces Bitcoin its proof is correct, or the Verifier wins, i.e. convinces Bitcoin the Prover's proof is incorrect. This framework can be applied to our motivating example: when Bob tries to withdraw, he is the Prover, with Larry being the Verifier. Bob wins if he succeeds in withdrawing, and Larry wins if he succeeds in stopping Bob from withdrawing. (We formalize the BitVM setting in Sec. 3 and use it to prove security in this paper.)

[1]In Satoshi Nakamoto's initial post introducing Bitcoin to the world, he described it as: "It's completely decentralized, without relying on central servers or trusted parties, because everything is based on crypto proofs rather than trust." [Nak09]

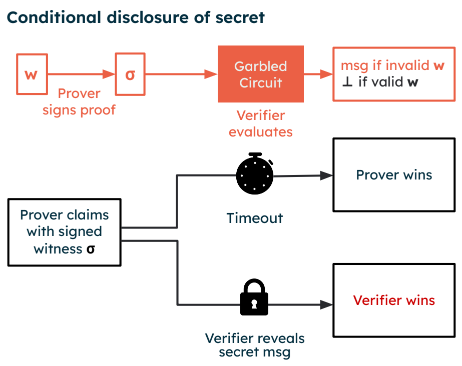


Figure 1: Verification on Bitcoin using a SNARK verifier garbled circuit. This is a solution to a cryptographic problem called conditional disclosure of secrets [GIKM00]. A Lamport signature σ serves both Bitcoin verification and as input labels to the garbled circuit.

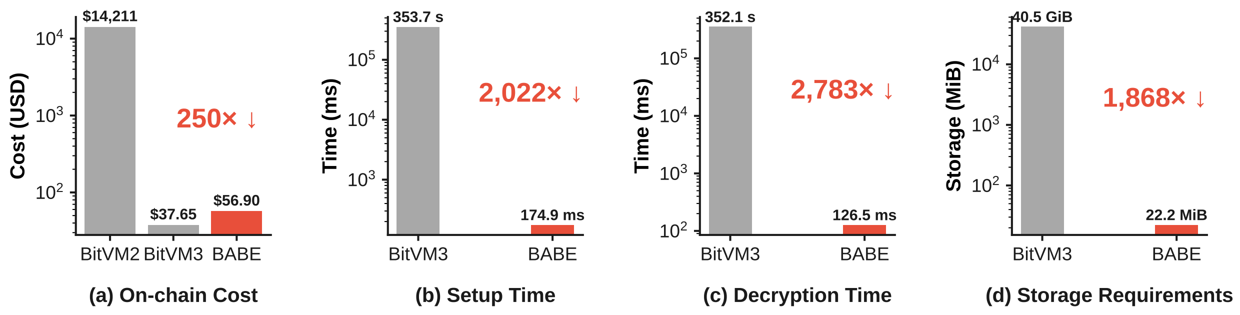
The state-of-the-art protocol in that line of work is BitVM2 [LAA+25]; this protocol follows the framework of naysayer proofs [SGB24] where the Prover posts the intermediate states of the proof verification computation trace on-chain and the Verifier can challenge a specific state transition by posting the correct verification trace only for that state transition. Even though this drastically reduces the on-chain costs as compared to posting the entire verification trace, the on-chain cost is still substantial as the intermediate states have to be signed by Lamport signatures [Lam79], verifiable on Bitcoin using hashlocks. (Lamport signatures are very large, requiring 256 bits per bit of each intermediate state.) In a mainnet experiment on Bitcoin [LAA+25], the on-chain footprint of posting these dispute transactions is over 5.4 Mbytes, costing over $\$15,000$ of transaction fees. This high challenge cost necessitates large amounts of capital locked as a bond to pay for the challenge. Moreover, the Bitcoin transactions involved in the challenge are non-standard[2] and require special services for submission to Bitcoin, thus increasing fee rates and centralization risks.

1.2.2 BitVM3

To reduce the on-chain costs of BitVM2, [Rub24] suggested using garbled circuits (GC) to shift the proof verification off-chain. Follow-up works include [Lin24, Che25, Eag25] under the general umbrella of BitVM3. Garbled circuits, originally invented by Yao [Yao82] for private two-party computation, are used in BitVM3 for *authenticated* computation. In the two-party setting, it works as follows (Fig. 1). At setup, the Prover generates a secret msg and shares a garbled circuit encrypting msg with the Verifier. The garbled circuit implements a SNARK verifier and can be used to decrypt the secret if an invalid proof is input into the garbled circuit. The Prover commits to the hashed value $H(\mathsf{msg})$ to create a hash lock such that the Verifier can open the hash lock and stop the Prover's claim if the Verifier can learn msg. The Verifier stores the garbled circuit. At the proving phase, the Prover posts a Lamport signed proof on Bitcoin, and the Verifier feeds the signed proof as input into the garbled circuit, with the Lamport signatures as input labels.[3] So if the proof is invalid, the Verifier will be able to stop the Prover by decrypting the scret and use it to open the hashlock. Otherwise, after a timeout, the Prover will win. This is basically a hash-time lock contract (HTLC) [Wik21] used in many Bitcoin protocols including atomic swaps [Wik20, Her18], Lightning [PD16], and Bitcoin staking [DLT+24]. HTLC had earlier been generalized to verification of

[2]Non-standard transactions are valid as per Bitcoin consensus rules but most miners do not consider them for inclusion in the blocks they mine because, for example, the transactions are too large.

[3]In an uncanny coincidence, Yao's input labels correspond exactly to a Lamport signature and vice-versa.



CPU: AMD Ryzen 7 7840U; timing: single-threaded. BTC price: $\$95,500$ (Jan 3, 2026). Fee rate: 2.2sats/vB for BitVM3, BABE, 11sats/vB for BitVM2 (premium rate for non-standard transactions). Details: Sec. 9.

Figure 2: On-chain and off-chain costs of BitVM2 [LAA+25], BitVM3 (Boolean garbled circuit for Groth16 verification) [Bit25b], and BABE. On-chain costs are derived from experiments on Bitcoin mainnet. These metrics are in the honest-setup setting where the Prover and Verifier are assumed to be honest during setup (but can be malicious afterwards). Sec. 9 evaluates multiple methods for achieving malicious security.

signatures by discreet log contracts [Dry17], and one can view BitVM3 as a final generalization to verification of arbitrary computations using the SNARK verifier garbled circuit.

The on-chain cost of BitVM3 compared to BitVM2 is reduced from $\$14,000$ to less than $\$40$ because only the signed proof has to be put on-chain instead of the signed intermediate states, and a hashlock script of negligible size replaces the verification traces. The challenge for BitVM3 is the significant *off-chain* costs. A Boolean garbled circuit for the Groth16 verifier [Gro16] was implemented [Bit25b], and even with free-XOR [KS08] and privacy-free half-gate [ZRE15] optimizations, the garbled circuit size is still 42 Gibytes (total of 10 B gates and 3 B non-free gates). Garbling time per circuit is 6 minutes on a single core. Moreover, proving the correctness of the garbled circuit with a zero-knowledge proof is infeasible and with a standard cut-and-choose method [LP07] requires garbling many instances of the circuits, translating to a total setup time of hours.

The large size of the garbled circuit for the Groth16 verifier primarily stems from the expensive pairing operations on the BN254 curve [BN06]. By replacing the Groth16 by a designated-verifier SNARK and by replacing the BN254 curve with a curve on the binary field, [Eag25] shows that the garbled circuit size can be reduced to 12 Mbytes. Unfortunately, the security of binary curves is not widely accepted by the community and so this approach is not currently pursued in practice. Arithmetic circuits were proposed recently [FBFL25] for garbling the Groth16 verifier, but the size is still expected to be 100 million ciphertexts (a few GBs).

1.3 New Verification Protocol: BABE

In this paper, we introduce a new verification protocol, BABE, which verifies Groth16 proofs (on the standard BN254 curve) on Bitcoin. BABE improves the off-chain costs of the existing Groth16 verifier garbled circuit [Bit25b] in BitVM3 by a factor of more than 1000 while preserving its low on-chain costs relative to BitVM2 (Fig. 2).

Instead of using a garbled circuit for the Groth16 verifier, our approach uses witness encryption [GGSW13] as a starting point. In simple terms, a witness encryption (WE) scheme for a relation R allows one to encrypt a secret under an NP statement $\mathbb{x} \in \mathcal{R}$ such that anyone that knows the corresponding witness $\mathbb{w}$ can decrypt.[4]

[4]Technically, this describes a stronger notion of witness encryption called "extractable witness encryption" [GKP+13]. For

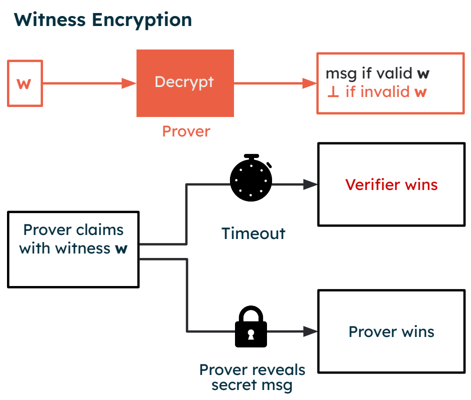


Figure 3: Verification on Bitcoin using witness encryption (if it were practical). Note that unlike under the conditional secret disclosure formulation, the secret is leaked when the proof is *valid*.

1.3.1 Witness Encryption

Let $\mathcal{R}_G$ be the relation for Groth16 verification, i.e. $\mathbb{x}$ is the statement (public inputs), $\mathbb{w}$ is the Groth16 proof (witness of this relation), and $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}_G$ is true iff the proof $\mathbb{w}$ is valid for the statement $\mathbb{x}$. If we have a solution for WE on $\mathcal{R}_G$, then we can use that to solve the Prover-Verifier problem above (Fig. 3).

The Verifier plays the role of the WE-encryptor. At setup, it encrypts a secret msg using the statement $\mathbb{x}$ to generate ciphertext ct. It creates a hash lock by committing to the hashed value $H(\mathsf{msg})$ of the secret msg. The Prover plays the role of the WE-decryptor and stores the ciphertext ct. At the proving phase, the Prover obtains the proof $\mathbb{w}$ of the application-specific relation, and runs $\mathsf{Dec}(\mathsf{ct}, \mathbb{w})$ to decrypt the secret msg. This is used to open the hash lock to win.

1.3.2 Witness encryption for linear pairings

Even before the advent of BitVM, witness encryption had already been proposed as a technique to perform trustless verification on Bitcoin (in the context of building Bitcoin bridges) [Hio22]. Unfortunately, witness encryption for all relations is a notoriously challenging cryptographic primitive, and to date it lacks an efficient realization for general relations.

Specifically, the Groth16 relation $\mathcal{R}_G$ involves verifying an equation containing pairing terms:

$$e(\pi_1, \pi_2) + e(\pi_3, x_1) = x_2 \tag{1}$$

Here, $\mathbb{w} = (\pi_1, \pi_2, \pi_3)$ is the proof, $\mathbb{x} = (x_1, x_2)$ is the statement. The issue is the pairing between the two witness elements ($e(\pi_1, \pi_2)$): There is no known efficient construction of WE for this type of relation.

However, we do know [BC16, GKPW24] an efficient WE scheme for relations that are pairing equations *linear* in the witness $\mathbb{w}$. Consider for example, the linear pairing relation $e(w_1, x_1) + e(x_2, w_2) = x_3$. Then we can obtain a WE scheme as follows:[5]

- **Relation**: $\mathcal{R}_{\text{linear}} = \{(\mathbb{x} = (x_1, x_2, x_3); \mathbb{w} = (w_1, w_2)) : e(w_1, x_1) + e(x_2, w_2) = x_3\}$
- **Encryption** (for $\mathbb{x}$): $\mathsf{ct} = (rx_1, rx_2, rx_3 + \mathsf{msg})$
- **Decryption** (using $\mathbb{w}$): $\mathsf{msg} = \mathsf{ct}_3 - e(\mathsf{ct}_2, w_2) - e(w_1, \mathsf{ct}_1)$

the sake of this overview we omit this delicate difference; we elaborate in the technical sections.

[5]Formally, for the security proof we need the "masking" term of the ciphertext to be passed through a random oracle, i.e. $\mathsf{RO}(rx_3)$. For simplicity, we omit this technicality from this overview and refer to Sec. 4.

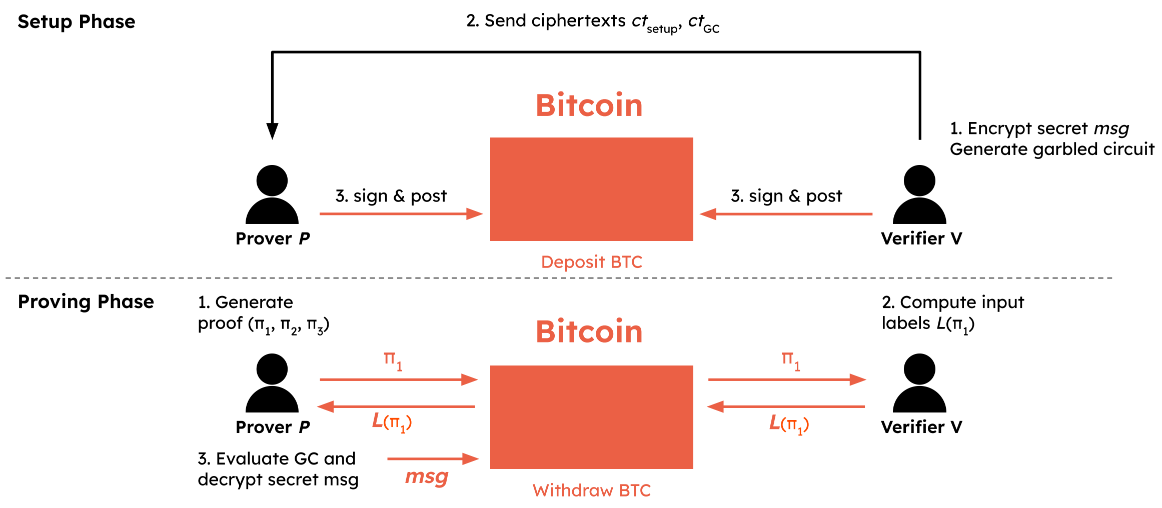


Figure 4: BABE: Verification of arbitrary computation on Bitcoin using a combination of linear witness encryption and an interactive protocol that allows the Prover to compute $r\pi_1$ without knowing r.

where r is an additional private random large field element generated by the WE-encryptor. This simple yet powerful observation has been leveraged to build efficient witness encryption schemes for specific relations across a wide range of cryptographic settings, both implicitly (e.g. [BL20, FKdP23, CFK24, FHAS24]) and explicitly following [GKPW24] (e.g. [CGPW25, AFP25, BFOQ25, MLLP25]), and has been recently formalized in [GHK+25]. To adapt this WE scheme for Groth16 verification, we treat one of the proof elements, π_1, as a public input x_0 of the WE relation so that the Groth16 relation (Eq. (1)) turns into:

$$e(x_0, \pi_2) + e(\pi_3, x_1) = x_2, \tag{2}$$

a linear pairing relation in the witness (π_2, π_3) and public inputs (x_0, x_1, x_2). Following the aforementioned framework for linear pairing equations, the ciphertext is given by:

$$\mathsf{ct} = (rx_0, rx_1, rx_2 + \mathsf{msg}). \tag{3}$$

But in reality, $x_0 := \pi_1$ is not available during setup, which means the ciphertext (Eq. (3)) cannot be entirely computed during setup; the computation must be deferred to the proving phase. In particular, the Prover must be able to compute $r\pi_1$ from π_1 in the proving phase without knowing the private randomness r of the Verifier.

To achieve this, the Prover and the Verifier will set up a maliciously secure two-party computation protocol (Fig. 4). This two-party computation takes as the Verifier's input the private randomness r at setup and as the Prover's input the proof element π_1 in the proving phase, and either the Prover learns $r\pi_1$ without learning r, or the protocol aborts and the Prover wins.

Concretely, during setup, the Verifier creates a garbled circuit ct_{GC} that hard codes the randomness r and shares it with the Prover. This garbled circuit computes $r\pi_1$ given input labels for π_1 with r private from the Prover. During the proving phase, when the Prover has the Groth16 proof, the Prover reveals π_1 on-chain and obtains the input label $L(\pi_1)$ from the Verifier. If the Verifier responds, then the Prover can compute $r\pi_1$ by inputting the labels into the garbled circuit. If the Verifier does not respond within a timeout or does not send the correct labels for the Prover's π_1, the protocol halts and the Prover wins.

Once the Prover has learned $r\pi_1$, together with $\mathsf{ct}_{setup} := (rx_1, rx_2 + \mathsf{msg})$ which it obtained from the Verifier at setup, it has the entire WE ciphertext (Eq. (3)). Now the Prover can use the rest of the proof (π_2, π_3) to decrypt the secret msg as in linear witness encryption and wins, provided that the proof is valid.

1.3.3 Garbled Circuit for Scalar Multiplication

In BitVM3, the garbled circuit (GC) allows authentic evaluation of a Groth16 verifier to reveal a secret. In BABE, the garbled circuit allows evaluation of a scalar multiplication $r\pi_1$ with privacy on r. So the garbled circuits play totally different roles in the two protocols. But more importantly, computing a single scalar multiplication is much simpler than computing a Groth16 verifier, which involves multiple pairings. More specifically, our goal is to design an efficient garbled circuit that

- takes as input labels a Lamport signature on $\pi_1 = (x, y) \in \mathbb{F}_p^2$ (Bitcoin friendly).
- Outputs $r\pi_1$.

A natural first approach is to leverage Yao's garbling [Yao82] for the scalar multiplication $r\pi_1$. While this approach offers a substantial $\approx 5\times$ improvement over the Groth16 verifier garbled circuit (BitVM3), the resulting solution would still be on the order of Gbytes. Essentially, the cost comes from the fact that one has to encode large ($\mathbb{F}_p$, where $\log(p) \approx 254$) field operations into binary circuits,[6] which introduces unnecessary overhead.

In this direction, a recent innovation [EL26] advocates an approach to keep most of the operations of the garbled circuit at the level of the relevant elliptic curve groups. Specifically, the paper introduces a new garbling primitive that computes vector homomorphic MACs (HMACs) on the BN254 elliptic curve. Such homomorphic MACs allow free additions of BN254 group elements, analogous to free-XORs for bits. The main result is an efficient garbling method to compute the components of the vector HMAC of a group element G, each of which effectively involves adding a fixed group element to G. The construction directly works over $\mathbb{F}_p$ and builds on the Ishai-Wee partial garbling for branching programs [IW14], which directly treats $\mathbb{F}_p$-elements without decomposing them to binary. Their main observation is that for BN254, a group addition to a fixed group element can be expressed as a low-degree polynomial f, such that $\pi_1 + \phi = f_\phi(x, y)$ where x, y are the x and y coordinates of π_1, (and thus a branching program) and garbled with [IW14]. Notably, the Ishai-Wee garbled circuit is information-theoretic and free, meaning that the garbled circuit has zero size, and the only cost comes from the encoded input (i.e. the labels). Crucially, the Ishai-Wee labels are not Bitcoin-friendly (i.e. Lamport signatures) as is the case for Yao. Therefore, the proposal suggests a pre-processing garbled circuit to compute the Ishai-Wee labels from the Lamport signature on π_1.

Building on this work, we develop a highly efficient garbling scheme for BN254 scalar multiplications. First, using the technique of decomposable randomized encodings [IK00, IK02, AIK04, Ish13] for linear operations, we decompose the BN254 scalar multiplication problem into the problem of computing a 254-dimensional vector HMAC. Second, to garble each component, we again rely on DRE for linear operations rather than the arguably more complex branching programs technique of Ishai-Wee used in [EL26]. Our core observation is that we can 'linearize' the BN254 group addition function $f_\phi(x, y)$ to $\widetilde{f_\phi}(x, y, x^2, y^2, xy)$, by simply giving low-degree monomials as inputs. Now our $\widetilde{f_\phi}$ is a linear function. Then the operation boils down to an inner-product, amenable to the [Ish13] DRE. The computation of the low-degree monomials is now deferred to the Yao garbled circuit, which turns out to be of low cost.

Comparing to [EL26], our technique is arguably simpler, allowing for a full security proof derived directly from the well-established decomposable randomized encodings for linear relations [FKN94, Ish13] (and a composition theorem within). Our final GC construction (after some optimizations) is about 22 MiB.

1.4 Other Applications

In addition to Bitcoin, several other blockchains (e.g. Cardano) do not natively support verification of general-purpose proofs. In our system BABE, the bulk of the protocol runs off-chain, and is therefore

[6] Recall that Yao's garbling works on binary circuits.

largely blockchain-agnostic. The on-chain component requires only very basic functionality including hashlocks, timelocks, and signature verification, which most blockchains already support. As a result, BABE or a system built on top of it, could be deployed across a broad class of chains, unlocking a wide range of applications beyond Bitcoin.

From a separate perspective, this work speaks to a broader usability bottleneck in cryptography: *non-black-box techniques* are often considered impractical because they require "opening up" cryptographic algorithms and expressing them as circuits, which can incur substantial overhead. A canonical example comes from zkSNARKs, where proving statements about cryptographic computations requires unrolling those computations into circuits, something that was historically seen as prohibitive. Over time, however, modern SNARK constructions and implementations have shown that these efficiency barriers can be overcome and that doing so can have far-reaching practical impact [BCTV14, OWWB20, BCMS20, BMM+21, GMN22, KST22, CFH+22, GGW24, OKMZ25, WOS+25, GGKS25].

Secure computation (e.g., via garbled circuits) faces a closely related challenge today. Despite significant progress, general-purpose garbling still performs poorly on many cryptographic workloads and other non-black-box computations, which remain expensive in practice. In this work, we focus on a highly specialized cryptographic task of real-world concern and demonstrate practical performance. Beyond the immediate application, this suggests a broader takeaway: with the right specialization, non-black-box uses of cryptography can be made practical for secure computation tasks, potentially enabling new deployments in a variety of settings.

2 Preliminaries

2.1 Basic Notation

We use λ for the security parameter and $\mathsf{negl}(\lambda)$ for a negligible function, i.e., a function that is less than $1/f(\lambda)$ for *any* polynomial f. We also define κ to be a statistical security parameter. The security parameters are implicitly taken as input to every algorithm and, for brevity, we omit explicitly writing it. For events A and B, we let $\Pr[A] \approx_\epsilon \Pr[B]$ denote that $|\Pr[A] - \Pr[B]| \leqslant \epsilon$. Row vectors will be written in small bold font, e.g. $\boldsymbol{x} = (x_1, \ldots, x_n)$ and matrices in capital bold, e.g. $\boldsymbol{A} = (a_{ij})_{i,j}$. We use the operator '$\times$' for the matrix multiplication and the operator $\otimes$ for the tensor product. $x \leftarrow\!\!\$\ \mathcal{X}$ is used to imply that x is being uniformly sampled from a finite set $\mathcal{X}$. "PPT" stands for Probabilistic Polynomial-Time. Every algorithm (including the adversaries and simulators) is stateful.

2.2 Bilinear Groups

A Bilinear Group $\mathcal{BG}$, generated as $(\mathbb{F}_q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, g_1, g_2, e) \leftarrow \mathcal{BG}(1^\lambda)$, is specified by a field $\mathbb{F}_q$ of prime order $q = 2^{\Theta(\lambda)}$, three groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ (the first two we call "source groups" and the third "target group"), a bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ that we call "pairing" and random generators g_1, g_2 for $\mathbb{G}_1, \mathbb{G}_2$ respectively. We use the *implicit notation*, i.e., $[x]_s := x g_s$ for $s \in \{1, 2, T\}$. Also, we denote the group operation additively, $[x]_s + [y]_s = [x + y]_s$, for $s \in \{1, 2, T\}$. The pairing has the property that $e([x]_1, [y]_2) = xye([1]_1, [1]_2)$. In our constructions, we will omit writing explicitly the Bilinear Group in the algorithms' inputs, even though it is implicitly taken as input.

2.2.1 Generic Bilinear Group Model

The Generic Group Model (GGM) [Sho97, Mau05] is an idealized model, that formalizes a 'generic' adversary. That is, the adversary does not have access to the concrete representation of the group elements but can only use generic group operations (addition, inverse element, scalar multiplication). This model captures the 'algebraic' attacks that an adversary can perform.

In this work, we use the Maurer's GGM [Mau05], which is extended to Bilinear Groups by [BBG05]. There, the adversary makes oracle queries for each generic group operation they wish to perform and receives a handle for the resulting group element, instead of the actual element itself. All group elements resulting from the adversary's queries are recorded—together with their handles—in three lists $\boldsymbol{L}_1, \boldsymbol{L}_2, \boldsymbol{L}_T$ for $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ respectively.

A standard GGM technique in security proofs is the 'symbolic' equivalence. We call 'symbolic' experiment (and symbolic group representation, respectively) the model where polynomials instead of group elements are stored and polynomial operations instead of group operations are performed. The formal variables of the polynomials are the initial elements that the adversary received. For example, a generic adversary to the discrete logarithm problem is initially receiving $[1], [x]$; thus, the formal variables are $1, X$, and then can perform any generic group operation which is going to be symbolically performed with the corresponding polynomials in $\mathbb{Z}_q[1, X]$.

Master Theorem. We recall the 'Master Theorem' [BBG05, Boy08]. The Master Theorem is used to determine the probability loss between the symbolic group representation and the actual generic group representation, where the formal variables are instantiated.

Theorem 1 (Master theorem [BBG05, Boy08]). *Let $\boldsymbol{L}_1 \in \mathbb{Z}_q[X_1, \ldots, X_n]^{\nu_1}$, $\boldsymbol{L}_2 \in \mathbb{Z}_q[X_1, \ldots, X_n]^{\nu_2}$, $\boldsymbol{L}_T \in \mathbb{Z}_q[X_1, \ldots, X_n]^{\nu_T}$ be three lists of n-variate polynomials over $\mathbb{Z}_q$ of maximum degree $d_{\boldsymbol{L}_1}, d_{\boldsymbol{L}_2}, d_{\boldsymbol{L}_T}$, respectively. Let $f \in \mathbb{Z}_q[X_1, \ldots, X_n]$ be an n-variate polynomial of degree d_f and denote $d = \max\{d_{\boldsymbol{L}_1} + d_{\boldsymbol{L}_2}, d_{\boldsymbol{L}_T}, d_f\}$, $\nu = \nu_1 + \nu_2 + \nu_T$. If f is* independent *of $(\boldsymbol{L}_1, \boldsymbol{L}_2, \boldsymbol{L}_T)$, then for any generic adversary $\mathcal{A}$ that makes at most q group oracle queries:*

$$\left| \Pr\left[\mathcal{A}\begin{pmatrix} p, \mathsf{h}_1[\boldsymbol{L}_1(\boldsymbol{x})], \\ \mathsf{h}_2[\boldsymbol{L}_2(\boldsymbol{x})], \mathsf{h}_T[f(\boldsymbol{x})] \end{pmatrix} = 1 \right] - \Pr\left[\mathcal{A}\begin{pmatrix} p, \mathsf{h}_1[\boldsymbol{L}_1(\boldsymbol{x})], \\ \mathsf{h}_2[\boldsymbol{L}_2(\boldsymbol{x})], \mathsf{h}_T[r] \end{pmatrix} = 1 \right] \right| \leqslant \frac{(q + \nu + 2)^2 d}{2p},$$

where $\mathsf{h}_1, \mathsf{h}_2, \mathsf{h}_T$ denote the corresponding handles, and the probabilities are taken over the choices of $\boldsymbol{x} \leftarrow\$ (\mathbb{Z}_q)^n$ and $r \leftarrow\$ \mathbb{Z}_q$.

Here f-dependence on $\boldsymbol{L} = (\boldsymbol{L}_1, \boldsymbol{L}_2, \boldsymbol{L}_T)$ means that the polynomial f is in the span of the polynomials in the list $\boldsymbol{C}(\boldsymbol{L}) := \{\boldsymbol{L}_1 \otimes \boldsymbol{L}_2\} \cup \boldsymbol{L}_T$ (intuitively $\{\boldsymbol{L}_1 \otimes \boldsymbol{L}_2\}$ are all the elements in $\mathbb{G}_T$ that can be computed using pairings). Naturally, the opposite case is called f-independence of $\boldsymbol{L}$.

2.3 Succinct Non-Interactive Arguments of Knowledge (SNARKs)

We recall the definition of SNARKs. In this work we do not focus on the zero-knowledge property, therefore we omit it.

Definition 1 (SNARKs). *A SNARK for a family of relations $\mathcal{R}_{\mathsf{fam}}$ is a tuple of three algorithms* $(\texttt{Gen}, \texttt{Prove}, \texttt{Verify})$*:*

- $\texttt{Gen}(\mathcal{R}) \to (\mathsf{crs})$*: On input a relation $\mathcal{R} \in R_{\mathsf{fam}}$ generates a common reference string* crs.
- $\texttt{Prove}(\mathsf{crs}, \boldsymbol{x}, \boldsymbol{w}) \to \pi$*: On input the common reference string* crs*, a statement $\boldsymbol{x}$ and the corresponding witness $\boldsymbol{w}$ computes a proof π.*
- $\texttt{Verify}(\mathsf{crs}, \boldsymbol{x}, \pi) \to 0/1$*: On input the common reference string* crs*, a statement $\boldsymbol{x}$ and a proof π the verification algorithm outputs either 1 for accept or 0 for reject.*

It is further required that the following properties hold.

(Perfect) Correctness. *For every relation $\mathcal{R} \in \mathcal{R}_{\mathsf{fam}}$, and every statement-witness pair $(\boldsymbol{x}, \boldsymbol{w}) \in \mathcal{R}$:*

$$\Pr\left[\ \mathtt{Verify}(\mathsf{crs}, \boldsymbol{x}, \pi) = 1 \ :\ \begin{array}{c} \mathsf{crs} \leftarrow \mathtt{Gen}(\mathcal{R}) \\ \pi \leftarrow \mathtt{Prove}(\mathsf{crs}, \boldsymbol{x}, \boldsymbol{w}) \end{array}\ \right] = 1$$

Knowledge Soundness. *For every PPT adversarial Prover $\mathcal{P}^*$, there exists a PPT extractor $\mathcal{E}$ such that for every benign auxiliary input* $\mathsf{aux} \in \{0,1\}^{\mathsf{poly}(\lambda)}$*, and every relation $\mathcal{R} \in \mathcal{R}_{\mathsf{fam}}$:*

$$\Pr\left[\ \begin{array}{l} \mathtt{Verify}(\mathsf{crs}, \boldsymbol{x}, \pi) = 1 \\ \wedge (\boldsymbol{x}, \boldsymbol{w}) \notin \mathcal{R} \end{array} \ :\ \begin{array}{c} \mathsf{crs} \leftarrow \mathtt{Gen}(\mathcal{R}) \\ (\boldsymbol{x}, \pi^*) \leftarrow \mathcal{P}^*(\mathsf{crs}, \mathsf{aux}) \\ \boldsymbol{w} \leftarrow \mathcal{E}(\mathsf{crs}, \mathsf{aux}) \end{array}\ \right] = \mathsf{negl}(\lambda)$$

Succinctness. *There exists a universal polynomial $p(\cdot)$ such that, for every security parameter $\lambda \in \mathbb{N}$, every relation $\mathcal{R} \in \mathcal{R}_{\mathsf{fam}}$, and every statement-witness pair $(\boldsymbol{x}, \boldsymbol{w})$:*

- *An honestly generated proof π has size $p(\lambda + \log|\boldsymbol{w}|)$.*
- *The Verifier algorithm $\mathtt{Verify}(\mathsf{crs}, \boldsymbol{x}, \pi)$ runs in time $p(\lambda + |\boldsymbol{x}| + \log|\boldsymbol{w}|)$.*

2.3.1 The Groth16 SNARK

We recall the Groth16 proof system [Gro16], excluding the zero-knowledge property.

Rank-1 constraint satisfiability (R1CS). Groth16 works for relations encoded with the rank-1 constraint satisfiability (R1CS). Formally, an R1CS relation is of the form:

$$\mathcal{R} = \left\{(\boldsymbol{x}; \boldsymbol{w}) : (\boldsymbol{A} \times \boldsymbol{z}^T) \circ (\boldsymbol{B} \times \boldsymbol{z}^T) = \boldsymbol{C} \times \boldsymbol{z}^T \wedge \boldsymbol{z} = (\boldsymbol{x} \| \boldsymbol{w}) \wedge x_1 = 1\right\}$$

where the relation is characterized by the fixed matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{Z}_p^{n \times m}$, the statement $\boldsymbol{x} \in \mathbb{Z}_p^{\ell}$ is an ℓ-sized vector,[7] the witness $\boldsymbol{w} \in \mathbb{Z}_p^{m-\ell}$ is an $(m-\ell)$-sized vector, and $\boldsymbol{z} \in \mathbb{Z}_p^m$ is called the 'extended witness', consisting of the witness and the statement. Here '$\circ$' is the Hadamard product. R1CS generalizes arithmetic circuits.

The Groth16 construction. For the proof system each column of the R1CS matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ is interpolated into polynomials as: $a_i(X) = \sum_{j=1}^n a_{i,j} L_j(X), b_i(X) = \sum_{j=1}^n b_{i,j} L_j(X), c_i(X) = \sum_{j=1}^n c_{i,j} L_j(X)$ for each $i \in [m]$, where $L_j(x)$ the corresponding Lagrange polynomial. Then the relation boils down to the following polynomial relation:

$$\left(\sum_{i=1}^m z_i a_i(X)\right)\left(\sum_{i=1}^m z_i b_i(X)\right) - \sum_{i=1}^m z_i c_i(X) = q(X)V(X)$$

where $V(X) = \prod_{i=1}^n (X - \omega^i)$ is the vanishing polynomial. This polynomial relation is essentially equivalent to the R1CS satisfiability (we refer to [GGPR13, PHGR13, Gro16] for more details).

The actual Groth16 SNARK is described below.

[7] In the R1CS encoding, the first entry of $\boldsymbol{x}$ is equal to 1 to rule out the degenerate case of the $\boldsymbol{z} = \boldsymbol{0}$ solution.

- $\texttt{Gen}(\mathcal{R}) \to \textsf{crs}$: Samples uniformly $\tau, \alpha, \beta, \gamma, \delta \leftarrow\!\$\ \mathbb{Z}_p$ and outputs:

$$\textsf{crs} = \Bigg\{ \{[\alpha]_1, [\beta]_2, [\gamma]_2, [\delta]_1, [\delta]_2, \left\{ \left[\tau^i\right]_1, \left[\tau^i\right]_2 \right\}_{i=0}^{n-1}, \left\{ \left[\frac{V(\tau)\tau^i}{\delta}\right]_1 \right\}_{i=0}^{n-2},$$
$$\Big\{[a_i(\tau)]_1, [b_i(\tau)]_1, [b_i(\tau)]_2\Big\}_{i=1}^{m}, \left\{ \left[\frac{\beta a_i(\tau) + \alpha b_i(\tau) + c_i(\tau)}{\gamma}\right]_1 \right\}_{i=1}^{\ell},$$
$$\left\{ \left[\frac{\beta a_i(\tau) + \alpha b_i(\tau) + c_i(\tau)}{\delta}\right]_1 \right\}_{i=\ell+1}^{m} \Bigg\}$$

- $\texttt{Prove}(\textsf{crs}, \boldsymbol{x}, \boldsymbol{w}, \pi) \to \pi$: Sets $\boldsymbol{z} = (\boldsymbol{x} \| \boldsymbol{w})$. Computes the quotient polynomial $q(X) = \sum_{i=0}^{n-2} q_i X^i = \left\lfloor \frac{(\sum_{i=1}^m z_i a_i(X))(\sum_{i=1}^m z_i b_i(X)) - \sum_{i=1}^m z_i c_i(X)}{V(X)} \right\rfloor$ and then outputs $\pi = (\pi_1, \pi_2, \pi_3)$, where:

$$\pi_1 = [\alpha]_1 + \sum_{i=1}^{m} z_i [a_i(\tau)]_1 \tag{4}$$

$$\pi_2 = [\beta]_2 + \sum_{i=1}^{m} z_i [b_i(\tau)]_2 \tag{5}$$

$$\pi_3 = \sum_{i=\ell+1}^{m} z_i \left[\frac{\beta a_i(\tau) + \alpha b_i(\tau) + c_i(\tau)}{\delta}\right]_1 + \sum_{i=0}^{n-2} \tilde{q}_i \left[\frac{V(\tau)\tau^i}{\delta}\right]_1 \tag{6}$$

- $\texttt{Verify}(\textsf{crs}, \boldsymbol{x}, \pi) \to 0/1$: Outputs 1 iff:

$$e(\pi_1, \pi_2) = e\left([\alpha]_1, [\beta]_2\right) + e\left(\sum_{i=1}^{\ell} x_i \left[\frac{\beta a_i(\tau) + \alpha b_i(\tau) + c_i(\tau)}{\gamma}\right]_1, [\gamma]_2 \right) + e(\pi_3, [\delta]_2) \tag{7}$$

Theorem 2 (Groth16 Security [Gro16]). *The above protocol is a SNARK, satisfying Perfect Correctness and Knowledge Soundness in the Generic Bilinear Group Model.*

2.4 Extractable Witness Encryption

We recall the definition of Extractable Witness Encryption [GKP+13]. Informally, an extractable witness encryption scheme for a relation $\mathcal{R}$ allows one to encrypt a message under an NP statement $\mathbb{x} \in \mathcal{R}$ such that anyone can decrypt if they *know* the corresponding witness $\mathbb{w}$, such that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$.[8]

Definition 2. *An Extractable Witness Encryption (WE) scheme for a relation $\mathcal{R}$ consists of three algorithms* $\textsf{WE} = (\texttt{Gen}, \texttt{Enc}, \texttt{Dec})$ *such that:*

- $\texttt{Gen}(\mathcal{R}) \to \textsf{crs}$*: Takes as input the relation and outputs a common reference string* $\textsf{crs}$.

- $\texttt{Enc}(\textsf{crs}, \mathbb{x}, \textsf{msg}) \to \textsf{ct}$*: Takes as input the common reference string* $\textsf{crs}$*, a statement* $\mathbb{x}$ *and a message* $\textsf{msg}$ *and outputs a ciphertext* $\textsf{ct}$.

- $\textsf{Dec}(\textsf{crs}, \textsf{ct}, \mathbb{w}) \to \textsf{msg}$*: Takes as input the common reference string, a ciphertext* $\textsf{ct}$ *and a witness* $\textsf{wit}$ *and outputs a message* $\textsf{msg}$.

Furthermore it should satisfy the following properties.

[8]In this paper, we denote the statement and witness used in a SNARK as $\boldsymbol{x}, \boldsymbol{w}$, and those used in witness encryption as $\mathbb{x}, \mathbb{w}$.

Correctness. *For every statement-witness pair* $(\mathbbm{x}, \mathbbm{w}) \in \mathcal{R}$ *and message* msg*:*

$$\Pr\left[\ \mathsf{Dec}(\mathsf{crs}, \mathsf{ct}, \mathbbm{w}) = \mathsf{msg} \ : \ \begin{array}{c} \mathsf{crs} \leftarrow \mathtt{Gen}(\mathcal{R}) \\ \mathsf{ct} \leftarrow \mathtt{Enc}(\mathsf{crs}, \mathbbm{x}, \mathsf{msg}) \end{array}\ \right] = 1 \tag{8}$$

Security. *An Extractable Witness Encryption for a relation* $\mathcal{R}$ *is secure if for every PPT adversary* $\mathcal{A}$*, there exists a PPT Extractor* $\mathcal{E}$ *such that for all benign auxiliary input* $\mathsf{aux} \in \{0,1\}^{\mathsf{poly}(\lambda)}$*, if*

$$\Pr\left[\ b = b' \ : \ \begin{array}{r} \mathsf{crs} \leftarrow \mathtt{Gen}(\mathcal{R}) \\ (\mathbbm{x}, \mathsf{msg}_0, \mathsf{msg}_1) \leftarrow \mathcal{A}(\mathsf{crs}) \\ b \leftarrow\!\!\$ \{0,1\} \\ \mathsf{ct}_b \leftarrow \mathtt{Enc}(\mathsf{crs}, \mathbbm{x}, \mathsf{msg}_b) \\ b' \leftarrow \mathcal{A}(\mathsf{ct}_b, \mathsf{aux}) \end{array}\ \right] = \frac{1}{2} + \epsilon \tag{9}$$

then

$$\Pr[(\mathbbm{x}, \mathbbm{w}) \in \mathcal{R} : \mathbbm{w} \leftarrow \mathcal{E}(\mathsf{crs}, \mathbbm{x}, \mathsf{aux})] \geq \epsilon - \mathsf{negl}(\lambda) \tag{10}$$

Remark 1 (Extractable WE vs (Plain) WE)**.** *Witness Encryption was originally introduced by Garg et al. [GGSW13] with a weaker security property that, roughly, the scheme is secure if* $\mathbbm{x} \notin \mathcal{R}$ *(i.e. there exists no* $\mathbbm{w}$ *such that* $(\mathbbm{x}, \mathbbm{w}) \in \mathcal{R}$*). In this work, we make use of the stronger version of extractable witness encryption, that can be secure even if* $\mathbbm{x} \in \mathcal{R}$ *(but the adversary doesn't know the witness). Throughout the paper, we, nevertheless, sometimes abuse the terminology and call it 'WE', omitting the 'extractable'.*

2.5 Garbling Schemes

Here we recall the definition of garbling schemes for binary circuits introduced by Yao in 1982 [Yao82] and further formalized in [LP09, BHR12b].

Definition 3. *A garbling scheme for a family of binary circuits* $\mathcal{C}_{\mathsf{fam}}$ *is a tuple of PPT algorithms* $\mathsf{GC} = (\mathsf{Garble}, \mathsf{Encode}, \mathsf{Eval})$ *such that:*

- $\mathsf{Garble}(C) \rightarrow (\mathsf{ct}_{\mathsf{GC}}, \mathsf{ek})$*: Takes as input a circuit* $C : \{0,1\}^n \rightarrow \{0,1\}^k \in \mathcal{C}_{\mathsf{fam}}$ *and outputs a garbled circuit description* $\mathsf{ct}_{\mathsf{GC}}$ *and an encoding key* ek*.*
- $\mathsf{Encode}(\mathsf{ek}, x) \rightarrow L_x$*: Takes as input an encoding key* ek *and a circuit input* $x \in \{0,1\}^n$ *and outputs the encoded input* L_x*. We also refer to* L_x *as the 'input labels' (or sometimes just 'labels').*
- $\mathsf{Eval}(\mathsf{ct}_{\mathsf{GC}}, L_x)$*: On input the garbled circuit* $\mathsf{ct}_{\mathsf{GC}}$ *and the labels of the input* L_x*, outputs the circuit output* $y \in \{0,1\}^k$

Furthermore, it should satisfy the following properties.

Correctness. *We require that for any binary circuit* $C \in \mathcal{C}_{\mathsf{fam}}$ *and any input* $x \in \{0,1\}^{|\mathsf{inp}(C)|}$*,*

$$\Pr\left[\ C(x) = \mathsf{Eval}(\mathsf{ct}_{\mathsf{GC}}, L_x) \ : \ \begin{array}{c} (\mathsf{ct}_{\mathsf{GC}}, \mathsf{ek}) \leftarrow \mathsf{Garble}(1^\lambda, C) \\ L_x \leftarrow \mathsf{Encode}(\mathsf{ek}, x) \end{array}\ \right] = 1$$

Adaptive Privacy. *We require that there exists a PPT simulator* $\mathsf{Sim} = (\mathsf{Sim}_1, \mathsf{Sim}_2)$ *such that for any PPT adversary* $\mathcal{A}$*, for every circuit* $C \in \mathcal{C}_{\mathsf{fam}}$ *and for every auxiliary information* $\mathsf{aux} \in \{0,1\}^{\mathsf{poly}(\lambda)}$*:*

$$\left| \Pr\left[\mathcal{A}(\mathsf{ct}_{\mathsf{GC}}, L_x, \mathsf{aux}) = 1 \;:\; \begin{array}{r} (\mathsf{ct}_{\mathsf{GC}}, \mathsf{ek}) \leftarrow \mathsf{Garble}(C) \\ x \leftarrow \mathcal{A}(\mathsf{ct}_{\mathsf{GC}}, \mathsf{aux}) \\ L_x \leftarrow \mathsf{Encode}(\mathsf{ek}, x) \end{array} \right] - \Pr\left[\mathcal{A}(\mathsf{ct}_{\mathsf{GC}}, L_x, \mathsf{aux}) = 1 \;:\; \begin{array}{r} (\mathsf{ct}_{\mathsf{GC}}, \mathsf{st}) \leftarrow \mathsf{Sim}_1(\mathsf{topo}(C)) \\ x \leftarrow \mathcal{A}(\mathsf{ct}_{\mathsf{GC}}, \mathsf{aux}) \\ L_x \leftarrow \mathsf{Sim}_2(\mathsf{st}, x, C(x)) \end{array} \right] \right| \leqslant \mathsf{negl}(\lambda)$$

The garbling scheme is called privacy-free *if it satisfies only correctness.*

In Yao's garbling scheme the encoding key is $\{L^b_{x,i}\}^n_{i=1,b=0,1}$ and the labels of x are $L_x = (L^{x_1}_{x,1}, \ldots, L^{x_n}_{x,n})$. Notice that fortuitously the algorithm coincide with a Lamport signature [Lam79] on x. Yao GC can be simply proven adaptively secure in the random oracle model [BHR12a]. We will make use of the well-established free-XOR [KS08] and half-gate [ZRE15] optimizations.

2.6 The Bitcoin Ledger

Ledger Model. We model Bitcoin as a distributed ledger functionality $\mathcal{F}_{\mathsf{BTC}}$ maintaining an append-only sequence of confirmed transactions (a *ledger*). We assume a global clock which counts time in rounds $r \in \mathbb{N}$ and $r \leqslant \mathsf{poly}(\lambda)$. Any party can submit a transaction tx to the ledger at any round r by calling $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx})$. For a party P and round $r \in \mathbb{N}$, the notation $\mathcal{L}^r_P$ denotes party P's local view of the Bitcoin ledger at round r.

The height of a ledger $\mathcal{L}$ is denoted $h(\mathcal{L})$. A ledger of height h is a sequence of blocks $\mathcal{L}[0], \ldots, \mathcal{L}[h-1]$. and $\mathcal{L}[: h]$ denotes the sequence of all blocks in $\mathcal{L}$ less than height h. $\mathcal{L}||B$ denotes the ledger formed by appending the block B to the ledger $\mathcal{L}$. Each block $\mathcal{L}[i]$ is a sequence of transactions. By flattening the blocks, the ledger itself can be viewed as a sequence of transactions. We denote by $\mathcal{L}_1 \preceq \mathcal{L}_2$ that $\mathcal{L}_1$ is a prefix of $\mathcal{L}_2$.

We assume the Bitcoin ledger satisfies the two fundamental properties safety and liveness. Informally, safety ensures that the ledgers of honest parties are consistent with each other, and liveness ensures that new valid transactions are eventually included in the ledgers of honest parties. Safety is defined as in [GKL15].

Definition 4 (Ledger Safety)**.** *A ledger functionality* $\mathcal{F}_{\mathsf{BTC}}$ *is* safe *if with probability* $1 - \mathsf{negl}(\lambda)$*: (i) for any honest party* P *and rounds* $r_1 \leq r_2$, $\mathcal{L}^{r_1}_P \preceq \mathcal{L}^{r_2}_P$ *(self-consistency); and (ii) for any honest parties* P_1, P_2 *and any round* r, $\mathcal{L}^r_{P_1} \preceq \mathcal{L}^r_{P_2}$ *or* $\mathcal{L}^r_{P_2} \preceq \mathcal{L}^r_{P_1}$ *(view-consistency)*

To precisely define liveness, we first define the validity of a Bitcoin transaction, beginning with a model for Bitcoin transactions. We discuss only the features of Bitcoin transactions that are relevant to BABE.

UTXO Model and Taproot Scripts. The ledger's state is represented as a set of unspent transaction outputs (UTXOs). Each UTXO is a pair $\mathsf{out} = (a, \mathsf{lockScript})$ where $a \in \mathbb{R}_{\geqslant 0}$ is the amount of coins (in BTC) in that UTXO, and lockScript is a program (the *locking script*) that determines under which conditions the UTXO can be spent.

In BABE, we widely use Taproot Trees [WNT20], or Taptrees, which make a UTXO spendable by satisfying one among multiple locking scripts. We will represent the locking script of such a UTXO as $\mathsf{lockScript} = \langle \mathsf{leaf}_0, \ldots, \mathsf{leaf}_{k-1} \rangle$.

Transactions. A transaction is a triple $\mathsf{tx} = (\mathsf{inputs}, \mathsf{tx\_witnesses}, \mathsf{outputs})$ where:

- $\mathsf{inputs} = [\mathsf{in}_1, \dots, \mathsf{in}_n]$, where each input references an output of a previous transaction, indexed as $\mathsf{in} = (\mathsf{PrevTx}, \mathsf{outIndex}, \mathsf{leaf})$ where PrevTx is the previous transaction, $\mathsf{outIndex}$ is the index of the output in the previous transaction, and leaf is the leaf of the Taproot tree to be satisfied.
- $\mathsf{outputs} = [\mathsf{out}_1, \dots, \mathsf{out}_m]$ where each output is a pair $(a, \mathsf{lockScript})$ as above;
- $\mathsf{tx\_witnesses} = [w_1, \dots, w_n]$, where w_i is a transaction witness (e.g., signature, data, etc.) intended to satisfy the corresponding input's leaf script.

The pair $\overline{\mathsf{tx}} = (\mathsf{inputs}, \mathsf{outputs})$ is called the *transaction skeleton*.

Cryptographic Primitives. We assume the existence of the following cryptographic primitives that are used by the ledger functionality:

- *Signature Scheme:* Sig_{BTC} with algorithms $\mathsf{Sig}_{BTC}.\mathtt{Gen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$, $\mathsf{Sig}_{BTC}.\mathsf{Sign}(\mathsf{sk}, \overline{\mathsf{tx}}) \to \sigma$, and $\mathsf{Sig}_{BTC}.\mathtt{Verify}(\mathsf{pk}, \overline{\mathsf{tx}}, \sigma) \to 0/1$ which is EUF-CMA secure.
- *Hash Function:* $\mathsf{Hash}_{BTC}(m) \to h$ modeled as a random oracle.
- *Lamport One-Time Signature Scheme [Lam79]:* $\mathsf{LampSig}$ with algorithms $\mathsf{LampSig}.\mathtt{Gen}(1^\lambda, \ell) \to (\mathsf{lsk}, \mathsf{lpk})$, $\mathsf{LampSig}.\mathsf{Sign}(\mathsf{lsk}, m) \to \mu$, and $\mathsf{LampSig}.\mathtt{Verify}(\mathsf{lpk}, m, \mu) \to 0/1$ where ℓ is the number of bits in the message m. A Lamport signature scheme constructed using Hash_{BTC} can be efficiently verified in Bitcoin script.

$$\mathsf{lsk} := \begin{pmatrix} L_0^0 & \dots & L_{\ell-1}^0 \\ L_0^1 & \dots & L_{\ell-1}^1 \end{pmatrix} \leftarrow\!\$ \left(\{0,1\}^\lambda\right)^{2\times\ell} \tag{11}$$

$$\mathsf{lpk} := \begin{pmatrix} \mathsf{Hash}_{BTC}(L_0^0) & \dots & \mathsf{Hash}_{BTC}(L_{\ell-1}^0) \\ \mathsf{Hash}_{BTC}(L_0^1) & \dots & \mathsf{Hash}_{BTC}(L_{\ell-1}^1) \end{pmatrix} \tag{12}$$

$$\mu := \begin{pmatrix} L_0^{m_0} & \dots & L_{\ell-1}^{m_{\ell-1}} \end{pmatrix} \tag{13}$$

$$\mathsf{LampSig}.\mathtt{Verify}(\mathsf{lpk}, m, \mu) = 1 \iff \forall i \in \{0, \dots, \ell-1\}, \mathsf{Hash}_{BTC}(\mu_i) = \mathsf{lpk}_i^{m_i} \tag{14}$$

Locking Scripts. The locking scripts of Bitcoin that are used by BABE are described below:

- *Signature Check:* $\mathsf{CheckSig}(\mathsf{pk})$ — Requires w to contain a digital signature σ such that $\mathsf{Sig}_{BTC}.\mathtt{Verify}(\mathsf{pk}, \overline{\mathsf{tx}}, \sigma) = 1$.
- *Relative Timelock:* $\mathsf{RelTimelock}(\tau)$ — Requires that at least τ blocks have elapsed since the output referenced by the input was created. That is, if a transaction with an input of the form $\mathsf{in} = (\mathsf{PrevTx}, \mathsf{outIndex}, \langle \mathsf{RelTimelock}(\tau) \rangle)$ is included in a block at height h, then PrevTx must have been included in a block at height at most $h - \tau$.
- *Hash Lock:* $\mathsf{HashLock}(h)$ — Requires that the transaction witness contains a hash pre-image of h, i.e. w such that $\mathsf{Hash}_{BTC}(w) = h$.
- *Logical Operations:* The above scripts can be combined using logical operator $\wedge$ (AND) and $\vee$ (OR) to form more complex locking conditions.

Using the above scripts and logical operator $\wedge$ (AND) and $\vee$ (OR), we can define more complex locking scripts, for example:

- *Lamport Signature:* $\mathsf{CheckLampSig}(\mathsf{lpk})$ — Requires that the transaction witness contains a Lamport signature for the public key lpk on some message. In BABE, we will use this script to ensure that the Prover, while posting this transaction, makes a binding commitment to some proof.

$$\mathsf{CheckLampSig}(\mathsf{lpk}) \ := \ \bigwedge_{i=1}^{\ell} \big(\mathsf{HashLock}(\mathsf{lpk}_i^0) \vee \mathsf{HashLock}(\mathsf{lpk}_i^1)\big) \tag{15}$$

- *Check Lamport Signatures from Multiple Parties on Same Message:* $\mathsf{CheckLampSigsMatch}(\mathsf{lpk}_A, \mathsf{lpk}_B)$ — Requires that the transaction witness contains a Lamport signature for each public key $\mathsf{lpk}_A, \mathsf{lpk}_B$, both on the same message. In BABE, we will use this script to ensure that the Verifier (who holds only lsk_B) commits to the same proof that the Prover (who holds lsk_A) committed to. This is guaranteed because the Verifier, who does not know lsk_A, cannot create a valid Lamport signature under lsk_A for any message other than the one committed to by the Prover. The Prover can then use the Verifier's commitment to evaluate the garbled circuit.

$$\begin{aligned}\mathsf{CheckLampSigsMatch}(\mathsf{lpk}_A, \mathsf{lpk}_B, \mathsf{msg}) \ := \ \bigwedge_{i=1}^{\ell} \Big[&\big(\mathsf{HashLock}((\mathsf{lpk}_A)_i^0) \wedge \mathsf{HashLock}((\mathsf{lpk}_B)_i^0)\big) \\ &\vee \big(\mathsf{HashLock}((\mathsf{lpk}_A)_i^1) \wedge \mathsf{HashLock}((\mathsf{lpk}_B)_i^1)\big)\Big]\end{aligned} \tag{16}$$

Definition 5 (Transaction Validity)**.** *A transaction* $\mathsf{tx} = (\mathsf{inputs}, \mathsf{tx\_witnesses}, \mathsf{outputs})$ *is* valid *with respect to a ledger* $\mathcal{L}$*, denoted* $\mathsf{Valid}_{\mathcal{L}}(\mathsf{tx})$*, if:*

1. All Inputs Unspent: *For each* $\mathsf{in}_i = (\mathsf{PrevTx}, \mathsf{outIndex}, \mathsf{leaf})$ *in* inputs*, the transaction* PrevTx *exists in* $\mathcal{L}$*.*

2. All Locking Scripts Satisfied: *Each input's specified leaf script is one of the leaves of the taptree for that input. That is, for each* $\mathsf{in}_i = (\mathsf{PrevTx}, \mathsf{outIndex}, \mathsf{leaf})$, $\mathsf{leaf} \in \langle \mathsf{leaf}_0, \ldots, \mathsf{leaf}_{k-1} \rangle =$ $\mathsf{PrevTx.outputs[outIndex].lockScript}$*. Moreover, the transaction witness* $\mathsf{tx\_witnesses}[i]$ *must satisfy the script* leaf*.*

3. Value Preservation: *The total amount of coins in the outputs is less than or equal to the total amount of coins in the inputs.*

$$\sum_{j=1}^{m} \mathsf{outputs}[j].a \leq \sum_{i=1}^{n} \mathsf{in}_i.\mathsf{PrevTx.outputs}[\mathsf{in}_i.\mathsf{outIndex}].a \tag{17}$$

A ledger $\mathcal{L}$ *is* valid*, denoted* $\mathsf{Valid}(\mathcal{L})$*, if for all transactions* tx *in* $\mathcal{L}$*,* $\mathsf{Valid}_{\mathcal{L}[:\mathsf{tx}]}(\mathsf{tx}) = 1$ *(*$\mathcal{L}[:\mathsf{tx}]$ *is the ledger containing all transactions in* $\mathcal{L}_P^r$ *before* tx*).*

Definition 6 (Ledger Validity)**.** *If a transaction* tx *appears in the ledger view* $\mathcal{L}_P^r$ *of any party* P *at any round* r*, then* tx *is valid with respect to the state* $\mathcal{L}_P^r[:\mathsf{tx}]$ *(the ledger containing all transactions in* $\mathcal{L}_P^r$ *before* tx*).*

Finally, to define liveness, we note that only valid transactions may be included in the ledger. However, the adversary may delay the inclusion of a valid transaction. The adversary may also include his own transactions in the ledger which may cause the honest party's transaction to become invalid thereafter, e.g. the adversary's transaction may use the same input as the honest party's transaction. Thus, liveness is guaranteed when the adversary is (computationally) unable to exclude a transaction for too many blocks.

To make this precise, we extend the definition of unambiguous transactions [GKL15] to *unstoppable transactions*. Informally, a transaction is unstoppable if no matter where in the next u blocks the adversary includes the honest party's transaction, this transaction will remain valid. For example, a transaction that requires the honest party's signature will be unstoppable since the adversary cannot forge the honest party's signature. In general, the adversary may have access to some state st, which may include certain signatures, ciphertexts, etc. shared by the honest party.

Definition 7 (u-Unstoppable Transactions)**.** *A transaction* tx *is* u-unstoppable *with respect to a ledger* $\mathcal{L}$ *and adversarial state* st*, if for all* PPT *adversaries* $\mathcal{A}$*:*

$$\Pr\left[\mathsf{Valid}(\mathcal{L}||B_1||\ldots||B_u) = 1 \ : \ \begin{array}{l} B_1, \ldots, B_u \leftarrow \mathcal{A}(\mathcal{L}, \mathsf{st}) \\ \exists\, i \in \{1, \ldots, u\} : \mathsf{tx} \in B_i \end{array}\right] \geqslant 1 - \mathsf{negl}(\lambda) \tag{18}$$

Definition 8 (Ledger Liveness)**.** *A ledger functionality* $\mathcal{F}_{\mathsf{BTC}}$ *is* u-live *(*$u \in \mathbb{N}$*) if for all adversarial states* st*, with probability* $1 - \mathsf{negl}(\lambda)$*, for any party* P *calling* $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx})$ *at any round* r *such that* tx *is* u*-unstoppable with respect to* $\mathcal{L}_P^r$ *and* st*, for all honest parties* H *and rounds* r' *with* $h(\mathcal{L}_H^{r'}) \geqslant h(\mathcal{L}_P^r) + u$*,* $\mathsf{tx} \in \mathcal{L}_H^{r'}[: h(\mathcal{L}_P^r) + u]$*.*

While the above definition of liveness guarantees inclusion of a transaction within a certain number of blocks, we use the *chain growth* property [GKL15] to guarantee that the height of every party's ledger grows.

Theorem 3 (τ-Chain Growth)**.** *For all* $s > \lambda$*, for all honest parties* P*,* $\Pr\left[\forall\, r : h(\mathcal{L}_P^{r+s}) < h(\mathcal{L}_P^r) + \tau s\right] \leqslant \mathsf{negl}(\lambda)$*.*

3 The BitVM-core Primitive

We define the BitVM-core primitive (c.f. BitVM2-core [LAA+25]).[9] A BitVM-core protocol is an interactive protocol run between a Prover P and a Verifier V interacting with the Bitcoin ledger functionality $\mathcal{F}_{\mathsf{BTC}}$. The protocol consists of two phases: a setup phase and a proving phase. In the setup phase, the Prover and Verifier interact off-chain to agree on the statement $\boldsymbol{x}$ to be proven and a set of Bitcoin transactions $\mathcal{T}$. A subset of these transactions $\mathcal{S}$ are the *withdraw transactions* because they pay Bitcoin to the Prover and can be posted by the Prover on Bitcoin if and only if he has a valid witness for the statement. During setup, the Prover and Verifier obtain and store local state (e.g., ciphertexts, private keys) to be used in the proving phase. To defend against malicious behavior, both the Prover and Verifier validate information received from the other party and may abort the setup if they detect invalid information.

During the proving phase, both the Prover and Verifier interact with the Bitcoin ledger. The Prover, who now knows a witness for the statement to be proven, and the Verifier take turns posting transactions from $\mathcal{T}$ to the ledger. In the end, the Prover wins by posting a withdraw transaction from $\mathcal{S}$ or the Verifier wins by preventing the Prover from ever posting a withdraw transaction. Security means that the Prover must win if he has a valid witness (even if the Verifier is malicious) and the Verifier must win otherwise (even if the Prover is malicious).

Protocol Syntax. A BitVM-core protocol for a relation $\mathcal{R}$ consists of one PPT algorithm $\mathtt{Gen}$ and four PPT interactive algorithms:

- $\mathtt{Gen}(\mathcal{R}) \rightarrow \mathsf{crs}$: Takes as input the relation and outputs a common reference string crs.

[9]Compared to BitVM2-core [LAA+25], we simplify the definition by considering a single Verifier and excluding the optimistic path. We discuss extensions in Sec. 8.

- $P_{\texttt{Setup}}(\mathsf{crs})$: Run by the Prover during the setup phase, interacts with the Verifier and the ledger functionality $\mathcal{F}_{\mathsf{BTC}}$, and outputs a statement $\boldsymbol{x}$, a set of transactions $\mathcal{T}$, a subset $\mathcal{S} \subseteq \mathcal{T}$, and the Prover's state st_P. Alternatively, the Prover may output $\perp$ to indicate that the setup failed.
- $V_{\texttt{Setup}}(\mathsf{crs})$: Run by the Verifier during the setup phase, interacts with the Prover and the ledger functionality $\mathcal{F}_{\mathsf{BTC}}$, and outputs a statement $\boldsymbol{x}$, a set of transactions $\mathcal{T}$, a subset $\mathcal{S} \subseteq \mathcal{T}$, and the Verifier's state st_V. Alternatively, the Verifier may output $\perp$ to indicate that the setup failed.
- $P_{\texttt{Prove}}(\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P, \boldsymbol{w})$: Run by the Prover knowing a witness $\boldsymbol{w}$, interacts with the ledger functionality $\mathcal{F}_{\mathsf{BTC}}$, and outputs a bit indicating successful proving.
- $V_{\texttt{Prove}}(\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_V)$: Run by the Verifier during the proving phase, interacts with the ledger functionality $\mathcal{F}_{\mathsf{BTC}}$.

We denote by $\langle P_{\texttt{Setup}}(\mathsf{crs}), V_{\texttt{Setup}}(\mathsf{crs})\rangle_r$ the random variable representing the transcript of the setup phase run interactively by the Prover and Verifier starting at round r (r is omitted when not relevant). The outputs of the Prover and Verifier during setup are denoted respectively by

$$\mathsf{out}_P(\langle P_{\texttt{Setup}}(\mathsf{crs}), V_{\texttt{Setup}}(\mathsf{crs})\rangle) = (\boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P) \text{ or } \perp \tag{19}$$

$$\mathsf{out}_V(\langle P_{\texttt{Setup}}(\mathsf{crs}), V_{\texttt{Setup}}(\mathsf{crs})\rangle) = (\boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_V) \text{ or } \perp. \tag{20}$$

and the union of both outputs is denoted by

$$\mathsf{out}\langle P_{\texttt{Setup}}(\mathsf{crs}), V_{\texttt{Setup}}(\mathsf{crs})\rangle = (\boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P, \mathsf{st}_V) \tag{21}$$

if neither party aborted, and $\perp$ otherwise. Similarly, the outputs of each party during the proving phase are denoted respectively by $\mathsf{out}_P(\langle P_{\texttt{Prove}}(\cdot), V_{\texttt{Prove}}(\cdot)\rangle)$ and $\mathsf{out}_V(\langle P_{\texttt{Prove}}(\cdot), V_{\texttt{Prove}}(\cdot)\rangle)$ where $(\cdot)$ is replaced by the inputs of the corresponding algorithm.

Robustness ensures that as long as the Prover is honest, accepted that the setup was performed correctly, and has a valid witness, a malicious Verifier cannot prevent the Prover from succeeding (i.e., the required withdraw transaction appears on the ledger) except with negligible probability. Thus, the worst thing a malicious Verifier can do is to cause the Prover to abort the setup, in which case no Bitcoin is ever lost by any party.

Definition 9 (u-Robustness). *For all NP relations $\mathcal{R}$, all* PPT *adversarial Verifiers V^*, all rounds $r \in \mathbb{N}$, the following holds:*

$$\Pr\left[\begin{array}{l} (b = 1) \\ \wedge\, (\exists\, \mathsf{tx} \in \mathcal{S} \text{ s.t.} \\ \mathsf{tx} \in \mathcal{L}_P^{r+u}) \end{array} : \begin{array}{l} \mathsf{crs} \leftarrow \mathtt{Gen}(\mathcal{R}) \\ (\boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P) \leftarrow \mathsf{out}_P(\langle P_{\texttt{Setup}}(\mathsf{crs}), V^*(\mathsf{crs})\rangle) \\ b \leftarrow \mathsf{out}_P(\langle P_{\texttt{Prove}}(\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P, \boldsymbol{w}), V^*(\mathsf{crs})\rangle_r) \\ (\boldsymbol{x}, \boldsymbol{w}) \in \mathcal{R} \end{array}\right] \geqslant \begin{array}{l} 1 - 2^{-\kappa} \\ -\,\mathsf{negl}(\lambda) \end{array} \tag{22}$$

Knowledge soundness ensures that if the Verifier accepts the setup, then a malicious Prover cannot successfully include the withdraw transaction on Bitcoin unless he knows a valid witness for the chosen relation. As usual in knowledge soundness, the Prover's knowledge of the witness is defined by the existence of an extractor that can extract the witness using the Prover's inputs and the transcript T of its interactions with the Verifier and Bitcoin.

Definition 10 (Knowledge Soundness). *For all NP relations $\mathcal{R}$, all* PPT *adversarial Provers P^*, there exists a* PPT *extractor $\mathcal{E}$ such that for every benign auxiliary input* $\mathsf{aux} \in \{0,1\}^{\mathsf{poly}(\lambda)}$*:*

$$\Pr\left[(\boldsymbol{x}, \mathcal{E}(\mathsf{crs}, T, \mathsf{aux})) \in \mathcal{R} : \begin{array}{l} \mathsf{crs} \leftarrow \mathtt{Gen}(\mathcal{R}) \\ (\boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_V) \leftarrow \mathsf{out}_V(\langle P^*(\mathsf{crs}, \mathsf{aux}), V_{\texttt{Setup}}(\mathsf{crs})\rangle) \\ T \leftarrow \langle P^*(\mathsf{crs}, \mathsf{aux}), V_{\texttt{Prove}}(\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_V)\rangle \\ \exists\, r \in \mathbb{N},\ \exists\, \mathsf{tx} \in \mathcal{S},\ \exists \text{ honest } H : \mathsf{tx} \in \mathcal{L}_H^r \end{array}\right] \geqslant \begin{array}{l} 1 - 2^{-\kappa} \\ -\,\mathsf{negl}(\lambda) \end{array} \tag{23}$$

where $\mathcal{E}$ above has the exact same view as the adversary.

Remark 2. *Assuming the existence of strong cryptographic primitives, there are auxiliary information distributions for which not all SNARKs admit an extractor [BCPR14, BP15]. Following the SNARK literature, we therefore, formally assume that the adversary should have access only to "benign" auxiliary inputs. Notably, these results are highly theoretical and do not affect SNARKs' security in practice.*

Together, robustness and knowledge soundness make a BitVM-core protocol *trustless*, solving the problem stated in Sec. 1. It is trustless because once the Prover and Verifier have mutually accepted that setup was performed correctly, an honest Prover can withdraw Bitcoin even if the Verifier is malicious, and an honest Verifier can prevent any malicious Prover from withdrawing Bitcoin.

Since a trivial protocol where the setup always aborts satisfies the above two properties, we also require that if both parties are honest, then neither party aborts during setup.

Definition 11 (Setup Correctness)**.** *For all NP relations $\mathcal{R}$,*

$$\Pr\left[\begin{array}{l}\mathsf{out}\langle P_{\mathtt{Setup}}(\mathsf{crs}), V_{\mathtt{Setup}}(\mathsf{crs})\rangle = \perp : \\ \quad \mathsf{crs} \leftarrow \mathtt{Gen}(\mathcal{R})\end{array}\right] \leqslant \mathsf{negl}(\lambda)$$

4 Witness Encryption for Linear Pairing Relation

In this section we present a core building block of our protocol, namely a Witness Encryption (WE) for Groth16, under the intermediate assumption that one proof element, π_1, is known to the Encryptor. Looking ahead, even though this is an unnatural assumption, combined with a garbling scheme it will constitute the cryptographic core of our full BABE construction.

Our crucial observation is that we can construct a simple and efficient Witness Encryption Scheme with respect to a Groth16 verification for an (R1CS) relation $\mathcal{R}$, a statement $\boldsymbol{x}$ and a specific proof element π_1.

Let any R1CS relation $\mathcal{R}$ and a Groth16 proof system for $\mathcal{R}$. Let $\mathsf{crs} \leftarrow \mathsf{Groth16.Gen}(\mathcal{R})$. We define the following relation:

$$\mathcal{R}' = \Big\{ \big((\mathsf{crs}, \boldsymbol{x}, \pi_1); \boldsymbol{w}\big) : \exists \pi_2, \pi_3 \text{ s.t. } \mathsf{Groth16.}\mathtt{Verify}(\mathsf{crs}, \boldsymbol{x}, (\pi_1, \pi_2, \pi_3)) = 1 \wedge (\boldsymbol{x}, \boldsymbol{w}) \in \mathcal{R} \Big\} \quad (24)$$

Construction 1 (WE for Groth16 with known π_1)**.** *Let $\mathcal{R}$ be an R1CS relation. Below is* $\mathsf{WE} = (\mathtt{Gen}, \mathtt{Enc}, \mathsf{Dec})$, *our witness encryption scheme for $\mathcal{R}'$:*
$\mathtt{Gen}(\mathcal{R}')$*: Output* $\mathsf{crs} = \mathsf{Groth16.Gen}(\mathcal{R})$
$\mathtt{Enc}(\mathsf{crs}, \boldsymbol{x}, \pi_1)$*: Samples $r \leftarrow\!\!\$\ \mathbb{F}_q^*$ and executes the two sub-algorithms:*

- $\mathtt{Enc}_{\mathsf{setup}}(\mathsf{crs}, \boldsymbol{x}, \mathsf{msg}; r)$*: Set* $Y := e\big([\alpha]_1, [\beta]_2\big) + e\big(X, [\gamma]_2\big)$ *and* $X := \sum_{i=1}^{\ell} x_i \left[\frac{\beta a_i(\tau) + \alpha b_i(\tau) + c_i(\tau)}{\gamma}\right]_1$ *and output* $\mathsf{ct}_{\mathsf{setup}} = (r[\delta]_2, \mathsf{RO}(rY) + \mathsf{msg})$
- $\mathtt{Enc}_{\mathsf{prove}}(\mathsf{crs}, \pi_1; r)$: *Output* $\mathsf{ct}_{\mathsf{prove}} = r\pi_1$
- *Outputs* $(\mathsf{ct}_{\mathsf{prove}}, \mathsf{ct}_{\mathsf{setup}})$

$\mathsf{Dec}(\mathsf{ct}, \boldsymbol{x})$*: Parse* $\mathsf{ct} := (\mathsf{ct}_{\mathsf{prove}}, \mathsf{ct}_{\mathsf{setup}}) = (\mathsf{ct}_1, (\mathsf{ct}_2, \mathsf{ct}_3))$. *Compute* $\pi_1, \pi_2, \pi_3 = \mathsf{Groth16.}\mathtt{Prove}(\mathsf{crs}, \boldsymbol{x}, \boldsymbol{w})$ *and output* $\mathsf{msg} = \mathsf{ct}_3 - \mathsf{RO}(e(\mathsf{ct}_1, \pi_2) - e(\pi_3, \mathsf{ct}_2))$.

Security of WE. In the following we show that this is effectively a witness encryption scheme for the relation $\mathcal{R}$, in the sense that the Decryptor cannot learn any information about msg unless she knows a valid witness $\boldsymbol{w}$ for the R1CS relation $\mathcal{R}$. In more detail, we show that WE is a secure witness encryption for $\mathcal{R}'$, therefore from a cheating adversary we can extract a valid witness $\boldsymbol{w}$ for the original relation $\mathcal{R}$.

In fact, we prove a stronger notion of security for our scheme, where an adversary adaptively chooses one part of the statement, π_1, after receiving a (partial) ciphertext, $\mathsf{ct}_{\mathsf{setup}}$ corresponding to $\boldsymbol{x}$. The type of adaptivity is not captured by the standard WE definition Def. 2; it is rather idiosyncratic to our scheme, as our encryption algorithm is explicitly split into two phases (setup and prove). We define this type of adaptive security for our scheme directly in Def. 12 below. Looking ahead, we will need this type of upgraded security for our full BABE protocol.

We prove security in the generic bilinear group model (see Sec. 2.2.1) and random oracle model. That means that the extractor has access to the adversary's queries to both oracles. The random oracle ensures that the extractor gets access to the group element that "randomizes" the message (here rY). Otherwise, from the point of view of the adversary the ciphertext perfectly hides the message msg. So, intuitively, the only way for the adversary to learn anything from the ciphertext is to query rY and learn the masking term. But then the extractor gets access to rY as well.

From there we use a standard generic group model argument to extract valid proof element π_2, π_3. Finally, we use the extractor of Groth16 to extract the original witness $\boldsymbol{w}$.

Remark 3. *As discussed in Sec. 1.3.2, our construction falls in the framework of witness encryption for general linear pairing relations introduced in [BC16, GKPW24] and subsequently formalized in [GHK⁺25]. Despite that, the security of our scheme (even the non-adaptive version) cannot be directly inferred from these prior works: [BC16] insists on relations in the standard model (does not capture Groth16), [GKPW24] identifies the general paradigm for arbitrary relations in the GGM, but provides security proof only for their particular instantiation and [GHK⁺25] formalizes the idea under a slightly different abstraction (linearly verifiable SNARKs). Consequently, we provide a complete security proof of our WE scheme for our specific relation.*

Definition 12 (Adaptive Security of WE for $\mathcal{R}'$)**.** *A witness encryption scheme* WE $= (\mathtt{Setup}, (\mathtt{Enc}_{\mathsf{setup}}, \mathtt{Enc}_{\mathsf{prove}}), \mathsf{Dec})$ *for $\mathcal{R}'$ is adaptively secure iff: For every PPT adversary $\mathcal{A}$, there exists a PPT Extractor $\mathcal{E}$ such that for all benign auxiliary input* $\mathsf{aux} \in \{0,1\}^{\mathsf{poly}(\lambda)}$, *if*

$$\Pr\left[b = b' \ : \ \begin{array}{r} \mathsf{crs} \leftarrow \mathtt{Gen}(\mathcal{R}) \\ (\boldsymbol{x}, \mathsf{msg}_0, \mathsf{msg}_1) \leftarrow \mathcal{A}(\mathsf{crs}) \\ b \leftarrow\!\$ \{0,1\}, r \leftarrow\!\$ \mathbb{F}_q^* \\ \mathsf{ct}^b_{\mathsf{setup}} \leftarrow \mathtt{Enc}_{\mathsf{setup}}(\mathsf{crs}, \boldsymbol{x}, \mathsf{msg}_b; r) \\ \pi_1 \leftarrow \mathcal{A}(\mathsf{ct}^b_{\mathsf{setup}}) \\ \mathsf{ct}_{\mathsf{prove}} \leftarrow \mathtt{Enc}_{\mathsf{prove}}(\mathsf{crs}, \pi_1; r) \\ b' \leftarrow \mathcal{A}(\mathsf{ct}_{\mathsf{prove}}, \mathsf{aux}) \end{array} \right] = \frac{1}{2} + \epsilon \tag{25}$$

then

$$\Pr[(\boldsymbol{x}, \boldsymbol{w}) \in \mathcal{R} : \boldsymbol{w} \leftarrow \mathcal{E}(\mathsf{crs}, \boldsymbol{x}, \mathsf{aux})] \geq \epsilon - \mathsf{negl}(\lambda) \tag{26}$$

Lemma 1. *Our* WE *construction for $\mathcal{R}'$ satisfies the adaptive security of Def. 12 in the generic bilinear group and random oracle models.*

Proof. Let a PPT adversary $\mathcal{A}$ of the above WE adaptive security game: $\mathsf{crs} \leftarrow \mathsf{WE}.\mathtt{Gen}(\mathcal{R}')$, chooses $(\boldsymbol{x}, \mathsf{msg}_0, \mathsf{msg}_1) \leftarrow \mathcal{A}(\mathsf{crs})$, $b \leftarrow\!\$ \{0,1\}$, $r \leftarrow\!\$ \mathbb{F}_q^*$, $\mathsf{ct}^b_{\mathsf{setup}} \leftarrow \mathsf{WE}.\mathtt{Enc}(\mathsf{crs}, \boldsymbol{x}, \mathsf{msg}_b; r)$, then chooses $\pi_1 \leftarrow \mathcal{A}(\mathsf{ct}^b_{\mathsf{setup}})$, $\mathsf{ct}_{\mathsf{prove}} \leftarrow \mathtt{Enc}_{\mathsf{prove}}(\mathsf{crs}, \pi_1; r)$ and finally outputs $b' \leftarrow \mathcal{A}(\mathsf{ct}_{\mathsf{prove}}, \mathsf{aux})$ such that $\Pr[b = b'] = \frac{1}{2} + \epsilon$ for an arbitrary ϵ.

We will construct an extractor $\mathcal{E}$ that on input crs and $\boldsymbol{x}$ (and the aux) outputs a valid witness $\boldsymbol{w}$, i.e. $\boldsymbol{w} \leftarrow \mathcal{E}(\mathsf{crs}, \boldsymbol{x}, \mathsf{aux})$, such that $\mathcal{R}(\boldsymbol{x}, \boldsymbol{w}) = 1$. Since we are in the generic group and random oracle models, $\mathcal{E}$ additionally has access to all the corresponding generic group and random oracle queries of $\mathcal{A}$.

First, we show that $\mathcal{A}$ queried the random oracle on rY with probability at least ϵ. Indeed

$$\begin{aligned}\frac{1}{2} + \epsilon = \Pr[b = b'] = &\Pr[b = b' | \text{“}rY \text{ Queried”}] \Pr[\text{“}rY \text{ Queried”}] \\ &+ \Pr[b = b' | \text{“}rY \text{ not Queried”}] \Pr[\text{“}rY \text{ not Queried”}]\end{aligned}$$

however if rY was not queried by $\mathcal{A}$ then $\mathsf{RO}(rY)$ information theoretically hides msg_b from $\mathcal{A}$, since $\mathsf{RO}(rY)$ is uniformly random. Consequently $\Pr[b = b' | \text{“}rY \text{ not Queried”}] = \frac{1}{2}$ which means that

$$\begin{aligned}\frac{1}{2} + \epsilon &= \Pr[b = b' | \text{“}rY \text{ Queried”}] \Pr[\text{“}rY \text{ Queried”}] + \frac{1}{2} \Pr[\text{“}rY \text{ not Queried”}] \\ &\leqslant \Pr[\text{“}rY \text{ Queried”}] + \frac{1}{2} \\ &\Rightarrow \Pr[\text{“}rY \text{ Queried”}] \geq \epsilon\end{aligned}$$

Now in the case where rY was queried by $\mathcal{A}$ we argue that, unless with negligible probability, $\mathcal{E}$ by observing the generic group oracle queries of $\mathcal{A}$ can extract a valid Groth16 proof $\pi = (\pi_1, \pi_2, \pi_3)$, i.e. $\mathsf{Groth16}.\mathtt{Verify}(\mathsf{crs}, \boldsymbol{x}, \pi) = 1$. We start from the symbolic group representation (see Sec. 2.2.1). Let $\mathfrak{r}$ be the symbolic variable of r, then information-theoretically the monomial $Y\mathfrak{r}$ can only be obtained from ct_1 and ct_2 queries, because no other element that the adversary receives contains the variable $\mathfrak{r}$. Therefore:

$$(Y)\mathfrak{r} = (a_2\pi_1)\mathfrak{r} + (a_3)\mathfrak{r}\mathfrak{d} \Rightarrow Y = a_2\pi_1 + a_3\mathfrak{d}$$

for a_2, a_3 chosen by the adversary ($\mathfrak{d}$ is the formal variable for δ).

Switching to the actual generic group model:

$$Y = e(\pi_1, [a_2]_2) + e([a_3]_1, [\delta]_2)$$

unless with probability $\mathsf{negl}_{\mathsf{GGM}}(\lambda) \leqslant \frac{(7+n+m)(m-\ell)}{2p} = \mathsf{negl}(\lambda)$ (this stems mostly from the Groth16 CRS; nm is the size of the R1CS matrices) determined by the Master Theorem (see Thm. 1). Recall that

$$Y := e\big([\alpha]_1, [\beta]_2\big) + e\Big(\sum_{i=1}^{\ell} x_i \left[\frac{\beta a_i(\tau) + \alpha b_i(\tau) + c_i(\tau)}{\gamma}\right]_1, [\gamma]_2\Big)$$

so $\pi_1, \pi_2 = [a_2]_2, \pi_3 = -[a_3]_1$ are valid Groth16 proof elements that $\mathcal{E}$ can compute by simply observing $\mathcal{A}$'s generic group oracle queries.

Finally, $\mathcal{E}$ invokes the knowledge soundness extractor of Groth16 on $\pi = (\pi_1, \pi_2, \pi_3) := (\pi_1, [a_2]_2, -[a_3]_1)$ and extracts a valid witness $\boldsymbol{w}$, unless with a negligible probability $\mathsf{negl}_{\text{Groth16-KS}}(\lambda)$.

In summary, the overall probability of success of $\mathcal{E}$ is $\epsilon - \mathsf{negl}_{\mathsf{GGM}}(\lambda) - \mathsf{negl}_{\text{Groth16-KS}}(\lambda) = \epsilon - \mathsf{negl}(\lambda)$.

□

Our adaptive security definition is stronger than the standard WE definition, therefore we get the following corollary.

Corollary 1. WE *is a secure witness encryption scheme for* $\mathcal{R}'$ *in the generic group and random oracle models.*

5 Garbled Circuit for BN254 Scalar Multiplication

5.1 Overview

Goal. The objective of this section is to construct a compact garbled circuit for BN254 scalar multiplication with a *hard-coded secret* scalar. Concretely, the garbling algorithm fixes a secret $r \in \mathbb{F}_q^*$, and the Prover provides a public input point $\pi \in \mathbb{G}$ (authenticated on-chain via a Lamport signature). The garbled evaluation outputs the group element $f_r(\pi) = r\pi$.

Input representation. The public input point $\pi = (x(\pi), y(\pi)) \in \mathbb{F}_p^2$ is supplied to the garbled circuit through labels corresponding to the bit-decomposition of its affine coordinates. The bit-decomposition is important because it enables verification of the labels on Bitcoin (see Eqs. (15) and (16)). From π, the circuit derives a small collection of algebraic features

$$\boldsymbol{u}(\pi) = \big(1,\ x(\pi),\ y(\pi),\ x(\pi)^2,\ y(\pi)^2,\ x(\pi)y(\pi)\big) \in \mathbb{F}_p^6,$$

and we also work with its bit-decomposition $\overline{\boldsymbol{u}}(\pi) \in \{0,1\}^\ell$ where $\ell = 1 + 5n$ and $n = \lceil \log_2 p \rceil$. The linear map $\boldsymbol{G} \in \mathbb{F}_p^{6\times\ell}$ reconstructs $\boldsymbol{u}(\pi)$ from $\overline{\boldsymbol{u}}(\pi)$, i.e., $\boldsymbol{u}^T(\pi) = \boldsymbol{G} \times \overline{\boldsymbol{u}}^T(\pi)$. Details in Sec. 5.2.

Observation: group addition becomes an inner product. A central bottleneck for naively garbling $r\pi$ is that scalar multiplication entails many elliptic-curve additions/doublings, which are prohibitively expensive inside a Boolean garbled circuit. Our first observation, as made by [EL26] is that, for BN254, addition of an input point $\pi \in \mathbb{G}$ to a fixed point $\phi \in \mathbb{G}$ (both in affine coordinates) can be expressed as *linear functions* of the feature vector $\boldsymbol{u}(\cdot)$. More precisely, assuming $x(\phi) \neq x(\pi)$, Lem. 3 shows that Jacobian coordinates of $\pi + \phi$ can be written as

$$(X, Y, Z) = \boldsymbol{A}(\phi) \times \boldsymbol{u}(\pi)^T,$$

for a matrix $\boldsymbol{A}(\phi) \in \mathbb{F}_p^{3\times 6}$ that depends only on ϕ. Since the Prover sees π only in bit form, we immediately lift this to the bit-decomposed domain using $\boldsymbol{G}$, yielding an equivalent form

$$(X, Y, Z) = \boldsymbol{B}(\phi) \times \overline{\boldsymbol{u}}(\pi)^T \qquad \text{where} \qquad \boldsymbol{B}(\phi) = \boldsymbol{A}(\phi) \times \boldsymbol{G} \in \mathbb{F}_p^{3\times\ell}.$$

Finally, because we will compute $\delta\pi + \phi$ where δ is a bit, we use the bit-gated variant (Lem. 4), which yields a matrix form $\boldsymbol{D}(\delta, \phi) \times \overline{\boldsymbol{u}}(\pi)^T$ for $\delta\pi + \phi$ where the coefficient matrix $\boldsymbol{D}$ depends only on (δ, ϕ).

Decomposable Randomized Encodings (DREs) To garble the resulting linear function $\boldsymbol{D}(\delta, \phi) \times \overline{\boldsymbol{u}}(\pi)^T$, we use a decomposable randomized encoding (DRE) for linear functions [Ish13]. A DRE for a function $f : X_1 \times \ldots \times X_n \to Y$ is a randomized encoding

$$\hat{f}(x = (x_1, \ldots, x_\ell), \rho) = (\hat{f}_1(x_1, \rho), \ldots, \hat{f}_\ell(x_\ell, \rho))$$

with randomness ρ such that there exists a decoder $\mathsf{Dec}(\hat{f}(x, \rho)) = f(x)$ but $\hat{f}(x, \rho)$ does not reveal any additional information about x (Defs. 13 and 14). Unlike garbled circuits, decoding a linear DRE requires no ciphertexts.

High-level plan. At a high level, we linearize the field operations (using the feature vector $\overline{\boldsymbol{u}}$), use the DRE decoding to compute all linear field- and group-level operations (which are expensive in a boolean circuit), and use the boolean garbled circuit for only low-degree non-linear operations:

1. Step 1: Create a *DRE for weighted group sum* to compute the scalar multiplication $r\pi$.
2. Step 2: Create another DRE to compute the DRE used in Step 1.
3. Step 3: Create a boolean garbled circuit for (i) validating the input is a valid Elliptic curve point, (ii) deriving the feature vector $\overline{\boldsymbol{u}}$, and (iii) computing the DRE used in Step 2.

Step 1: DRE for the weighted group sum with public weights. To efficiently do scalar multiplication, we take $r = \sum_{i=0}^{n-1} r_i 2^i$, the binary expansion of the hard-coded scalar $r \in \mathbb{F}_q^*$. Using the DRE for weighted summation in abelian groups (Lem. 8), we obtain a DRE for the function

$$f(\, r_0\pi, \ldots, r_{n-1}\pi \,) = \sum_{i=0}^{n-1} 2^i (r_i \pi) = r\pi.$$

The DRE for this function is

$$\hat{f}(\, r_0\pi, \ldots, r_{n-1}\pi \,) = (r_0\pi + \rho_0, \ldots, r_{n-1}\pi + \rho_{n-1}\,)$$

where $\rho_0, \ldots, \rho_{n-1}$ are sampled uniformly from $\mathbb{G}_1 \setminus \{\mathcal{O}\}$ subject to $\sum_{i=0}^{n-1} 2^i \rho_i = 0$.[10] This DRE shifts this *weighted summation* (here $2^0, \ldots, 2^{n-1}$ are the weights) from the circuit to the Prover: the Prover simply computes the weighted sum of the elements of $\hat{f}$ to obtain $r\pi_1$.

Step 2: DRE for producing each $r_i\pi + \rho_i$ from $\overline{\boldsymbol{u}}(\pi)$. This step creates a DRE to compute $r_i\pi + \rho_i$ for random masks ρ_i used in Step 1. Here we use Lem. 4 to express $r_i\pi + \rho_i$ in Jacobian coordinates as a linear map of $\overline{\boldsymbol{u}}(\pi)$:

$$(X_i, Y_i, Z_i) \;=\; \boldsymbol{D}(r_i, \rho_i) \times \overline{\boldsymbol{u}}(\pi)^T,$$

for a matrix $\boldsymbol{D}(r_i, \rho_i) \in \mathbb{F}_p^{3\times\ell}$ depending only on the private inputs (r_i, ρ_i). Each of the three coordinates is thus an inner product between a *known* coefficient vector and the *bit-vector* $\overline{\boldsymbol{u}}(\pi)$. We then apply a DRE for private affine functions (Lem. 9) to create a DRE for these inner products. The DRE takes the form

$$\hat{h}_k(r, \overline{u}_k, \omega) = \{\overline{u}_k \cdot D_{i,j,k} + s_{i,j,k}\}_{i\in\{0,\ldots,n-1\}, j\in\{1,2,3\}} \quad \text{for} \quad k \in \{1, \ldots, 5n+1\}$$

where $D_{i,j,k}$ is the (j,k)-th entry of $\boldsymbol{D}(r_i, \rho_i)$ as defined above, and ω represents all the randomness used in the DRE, including ρ_i from Step 1 and $s_{i,j,k}$ which are uniformly random subject to $\sum_{k=1}^{5n+1} s_{i,j,k} = 0$ for all $i \in \{0, \ldots, n-1\}$, $j \in \{1, 2, 3\}$. The complete DRE construction is given in Thm. 4.

Step 3: The Boolean Circuit. The garbled circuit is formally defined in Constr. 2.

Garbling: Outputs a Lamport secret key (the encoding key)

$$\mathsf{ek} = \begin{pmatrix} L_{x,0}^0 & \cdots & L_{x,n-1}^0 & L_{y,0}^0 & \cdots & L_{y,n-1}^0 \\ L_{x,0}^1 & \cdots & L_{x,n-1}^1 & L_{y,0}^1 & \cdots & L_{y,n-1}^1 \end{pmatrix}$$

and the garbled circuit $\mathsf{ct}_{\mathsf{GC}}$ computed in three steps:

1. **Elliptic curve validation:** The garbled circuit first verifies that the input point $\pi = (x(\pi), y(\pi))$ lies on the curve E, i.e., it checks that $y(\pi)^2 \equiv x(\pi)^3 + 3 \pmod p$.

[10] Note that in the actual construction, we also need to randomize the representation on Jacobian coordinates. We refer the reader to Remark 4 and Lem. 5 for more details.

2. **Binary decomposition:** If the validation passes, the circuit computes the binary decomposition $\overline{\boldsymbol{u}}(\pi) \in \{0,1\}^{1+5n}$ of $\boldsymbol{u}(\pi) = (1, x(\pi), y(\pi), x(\pi)^2, y(\pi)^2, x(\pi)y(\pi))$. These two steps are implemented using a privacy-free Boolean garbled circuit. Let $L_{\overline{u},k}^{\overline{u}_k(\pi)}$ for $k \in [\ell]$ be the output labels of this step.

3. **DRE encoding:** For each $k \in \{1, \ldots, \ell\}$ where $\ell = 1 + 5n$, the garbled circuit comes hardwired with the DRE encoding from Thm. 4. Specifically, the garbled circuit contains an encryption of $\hat{h}_k((r,0),\omega)$ under the 0-label $L_{\overline{u},k}^0$ and an encryption of $\hat{h}_k((r,1),\omega)$ under the 1-label $L_{\overline{u},k}^1$ for $\overline{u}_k(\pi)$.

Evaluation: The evaluator, given the input labels

$$L = \left(L_{x,0}^{x(\pi)_0} \quad \ldots \quad L_{x,n-1}^{x(\pi)_{n-1}} \quad L_{y,0}^{y(\pi)_0} \quad \ldots \quad L_{y,n-1}^{y(\pi)_{n-1}} \right),$$

evaluates the Boolean circuits to obtain labels for $\overline{u}(\pi)$, and decrypts the appropriate encryptions to obtain $\hat{h}_k(r, \overline{u}_k(\pi))$ for each k. The evaluator then decodes the DRE as follows (Thm. 4):

- Recast $h_k(r, \overline{u}_k(\pi), \omega)$ as $\{t_{i,j,k}\}_{i \in \{0,\ldots,n-1\}, j \in \{1,2,3\}}$. Then the Prover aggregates shares to cancel the $s_{i,j,k}$-terms: $\widehat{t}_{i,j} = \sum_{k=1}^{5n+1} t_{i,j,k}$.
- Interpreting $(\widehat{t}_{i,1}, \widehat{t}_{i,2}, \widehat{t}_{i,3})$ as the Jacobian coordinates of a masked point Q_i, the final output is obtained by the usual bit-weighted recombination out $= \sum_i 2^i Q_i$. Since $Q_i = r_i\pi + \rho_i$ and $\sum_i 2^i \rho_i = 0$, then out $= r\pi$.

Intuitively, the heavy work (group operations and most algebra) is pushed to the Prover and to the Verifier's offline preprocessing, while the online garbled circuit mainly performs validation, low-level arithmetic for feature extraction, and symmetric-key decryptions for table selection to output the precomputed DRE components.

5.2 Elliptic Curve Addition and Notation

Let $E/\mathbb{F}_p$ be the BN254 curve in short Weierstrass form

$$E : y^2 = x^3 + 3 \pmod{p},$$

where p is the BN254 base-field prime. Let $n = \lceil \log_2 p \rceil$ denote the bit length of $\mathbb{F}_p$ elements (so $x(\pi)$, $y(\pi)$ have n bits each). For BN254 we also have $n = \lceil \log_2 q \rceil$, the bit length of $\mathbb{F}_q^*$, where q is the BN254 group order.

Let π be a point on the curve, represented in affine coordinates $\pi := (x(\pi), y(\pi))$ with $x(\pi), y(\pi) \in \mathbb{F}_p$. Then we define $\boldsymbol{u}(\pi) = (1, x(\pi), y(\pi), x(\pi)^2, y(\pi)^2, x(\pi)y(\pi))$ and $\overline{\boldsymbol{u}}(\pi) \in \{0,1\}^\ell$ be the binary decomposition of $\boldsymbol{u}(\pi)$, where $\ell = 5n + 1$. More precisely, if $\boldsymbol{u}(\pi) = (u_0, u_1, u_2, u_3, u_4, u_5)$ where $u_0 = 1$, $u_1 = x(\pi)$, $u_2 = y(\pi)$, $u_3 = x(\pi)^2$, $u_4 = y(\pi)^2$, and $u_5 = x(\pi)y(\pi)$, then:

$$\overline{\boldsymbol{u}}(\pi) = (1, u_{1,0}, \ldots, u_{1,n-1}, u_{2,0}, \ldots, u_{2,n-1}, u_{3,0}, \ldots, u_{3,n-1}, u_{4,0}, \ldots, u_{4,n-1}, u_{5,0}, \ldots, u_{5,n-1})$$

where $(u_{i,0}, \ldots, u_{i,n-1})$ is the binary decomposition of u_i for $i \in \{1,2,3,4,5\}$ (i.e., $u_i = \sum_{j=0}^{n-1} u_{i,j} 2^j$). We use the notation $u_i(\pi)$ and $\overline{u}_i(\pi)$ to denote the i^{th} component of $\boldsymbol{u}(\pi)$ and $\overline{\boldsymbol{u}}(\pi)$, respectively.

Further, we define $\boldsymbol{G} \in \mathbb{F}_p^{6 \times (1+5n)}$ such that $\boldsymbol{G} \times \overline{\boldsymbol{u}}^T(\pi) = \boldsymbol{u}^T(\pi)$. The matrix $\boldsymbol{G}$ is structured as:

$$\boldsymbol{G} = \left(\begin{array}{c|ccccc} 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \hline 0 & \mathbf{2} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 0 & \mathbf{0} & \mathbf{2} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 0 & \mathbf{0} & \mathbf{0} & \mathbf{2} & \mathbf{0} & \mathbf{0} \\ 0 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{2} & \mathbf{0} \\ 0 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{2} \end{array} \right)$$

where $\mathbf{0}$ denotes a row vector of n zeros, and $\mathbf{2}$ denotes the row vector $(2^0, 2^1, \ldots, 2^{n-1})$ of powers of 2. That is, the first column of $\boldsymbol{G}$ is $(1, 0, 0, 0, 0, 0)^T$ (corresponding to the constant term $u_0 = 1$), and for each $i \in \{1, 2, 3, 4, 5\}$, columns $(i-1)n+2$ through $in+1$ contain the powers of 2 $(2^0, 2^1, \ldots, 2^{n-1})$ in row $i+1$, with zeros elsewhere. This ensures that $\boldsymbol{G} \times \overline{\boldsymbol{u}}^T(\pi)$ reconstructs the field elements from their binary decomposition.

The following lemmata Lem. 2, Lem. 3, Lem. 4, Lem. 5 formally define the elliptic curve addition and, further, $\phi + \delta\pi$ operation in Jacobian coordinates.

Lemma 2. *Let $\pi = (x(\pi), y(\pi)) \in E(\mathbb{F}_p) \setminus \{\mathcal{O}\}$ be given in affine coordinates, and let $\delta \in \{0, 1\}$. Define $\boldsymbol{M}(\delta) \in \mathbb{F}_p^{3\times 6}$ by*

$$\boldsymbol{M}(\delta) = \begin{pmatrix} 1-\delta & \delta & 0 & 0 & 0 & 0 \\ 1-\delta & 0 & \delta & 0 & 0 & 0 \\ \delta & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Then $\boldsymbol{M}(\delta) \times \boldsymbol{u}^T(\pi)$ is a Jacobian representation of the point $\delta\pi$: when $\delta = 1$ it equals $(x(\pi), y(\pi), 1)$, and when $\delta = 0$ it equals $(1, 1, 0)$, which represents $\mathcal{O}$ under the convention that any $(X, Y, 0)$ with $Y \neq 0$ encodes the point at infinity. Alternatively, $\boldsymbol{N}(\delta) \times \overline{\boldsymbol{u}}^T(\pi)$ is also equal to $\delta\pi$, where $\boldsymbol{N}(\delta) = \boldsymbol{M}(\delta) \times \boldsymbol{G}$.

The proof of this lemma is straightforward and is left to the reader.

Lemma 3. *Let $\phi, \pi \in E(\mathbb{F}_p) \setminus \{\mathcal{O}\}$ be two points in affine coordinates such that $x(\phi) \neq x(\pi)$. Then a Jacobian representation of $\phi + \pi$ is given by $\boldsymbol{A}(\phi) \times \boldsymbol{u}^T(\pi)$ where $\boldsymbol{A}(\phi) \in \mathbb{F}_p^{3\times 6}$ is a matrix that depends only on ϕ, defined by:*

$$\boldsymbol{A}(\phi) = \begin{pmatrix} 6 & x^2(\phi) & -2y(\phi) & x(\phi) & 0 & 0 \\ 9y(\phi) & 0 & -(y(\phi)^2 + 9) & 3x(\phi)y(\phi) & y(\phi) & -3x^2(\phi) \\ -x(\phi) & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Equivalently, Jacobian coordinates of $\phi + \pi$ are also given by $\boldsymbol{B}(\phi) \times \overline{\boldsymbol{u}}^T(\pi)$ where $\boldsymbol{B}(\phi) \in \mathbb{F}_p^{3\times(1+5n)}$ is defined as $\boldsymbol{B}(\phi) = \boldsymbol{A}(\phi) \times \boldsymbol{G}$.

Proof. Let $\phi = (x_1, y_1)$ and $\pi = (x_2, y_2)$ be affine points on

$$E/\mathbb{F}_p : \quad y^2 = x^3 + 3,$$

and assume $x_1 \neq x_2$. Define

$$Z \overset{\text{def}}{=} x_2 - x_1 \in \mathbb{F}_p^*.$$

Consider the following projective (Jacobian-style) representation of $\phi + \pi$:

$$\begin{aligned} X_3 &\overset{\text{def}}{=} (y_2 - y_1)^2 - (x_1 + x_2)\, Z^2, \\ Y_3 &\overset{\text{def}}{=} (y_2 - y_1)\,(x_1 Z^2 - X_3) - y_1 Z^3, \\ Z_3 &\overset{\text{def}}{=} Z. \end{aligned} \tag{27}$$

When $Z \neq 0$, the triple $(X_3 : Y_3 : Z_3)$ represents the affine point

$$\left(\frac{X_3}{Z_3^2}, \frac{Y_3}{Z_3^3} \right)$$

and one checks that it equals $\phi + \pi$ (this is the standard affine-addition formula written in projective coordinates; see, e.g., the explicit projective addition formulas for short Weierstrass curves).

We now expand (27) and simplify using the curve equations

$$y_1^2 = x_1^3 + 3, \qquad y_2^2 = x_2^3 + 3.$$

Derivation of Z_3. By definition, $Z_3 = Z = x_2 - x_1$, which is already linear in $\boldsymbol{u}(\pi)$.

Derivation of X_3. First expand:

$$\begin{aligned} X_3 &= (y_2 - y_1)^2 - (x_1 + x_2)(x_2 - x_1)^2 \\ &= (y_2^2 - 2y_1y_2 + y_1^2) - (x_1 + x_2)(x_2^2 - 2x_1x_2 + x_1^2) \\ &= y_2^2 - 2y_1y_2 + y_1^2 - \left(x_1^3 - x_1^2x_2 - x_1x_2^2 + x_2^3\right). \end{aligned}$$

Substitute $y_1^2 = x_1^3 + 3$ and $y_2^2 = x_2^3 + 3$ to obtain cancellation of the x_1^3 and x_2^3 terms:

$$X_3 = (x_2^3 + 3) - 2y_1y_2 + (x_1^3 + 3) - x_1^3 + x_1^2x_2 + x_1x_2^2 - x_2^3,$$

hence

$$X_3 = 6 + x_1^2x_2 - 2y_1y_2 + x_1x_2^2. \tag{28}$$

Derivation of Y_3. Start from (27):

$$Y_3 = (y_2 - y_1)\left(x_1Z^2 - X_3\right) - y_1Z^3.$$

Using $Z = x_2 - x_1$, expand and simplify to:

$$Y_3 = y_1x_2^3 - 3x_1^2x_2y_2 + 3x_1x_2^2y_1 - y_1^2y_2 + y_1y_2^2 + 3(y_1 - y_2).$$

(One can obtain this by a straightforward polynomial expansion and collecting terms.) Now use the curve relations to eliminate the cubic term: since $y_2^2 = x_2^3 + 3$, we have $x_2^3 = y_2^2 - 3$, so

$$y_1x_2^3 = y_1(y_2^2 - 3) = y_1y_2^2 - 3y_1.$$

Substituting this gives:

$$\begin{aligned} Y_3 &= (y_1y_2^2 - 3y_1) - 3x_1^2x_2y_2 + 3x_1x_2^2y_1 - y_1^2y_2 + y_1y_2^2 + 3y_1 - 3y_2 \\ &= -3x_1^2x_2y_2 + 3x_1x_2^2y_1 - y_1^2y_2 + 2y_1y_2^2 - 3y_2. \end{aligned}$$

Finally, rewrite $-3y_2$ as $9y_1 - (y_1^2 + 9)y_2$ plus terms that cancel using $y_1^2 = x_1^3 + 3$ (equivalently, just regroup to match the $u(\pi)$ basis). A clean regrouping yields:

$$Y_3 = 9y_1 \; - \; (y_1^2 + 9)\, y_2 \; + \; 3x_1y_1\, x_2^2 \; + \; y_1\, y_2^2 \; - \; 3x_1^2\, (x_2y_2). \tag{29}$$

Matrix form. Putting together the above, we have that $\boldsymbol{A}(\phi) \times \boldsymbol{u}^T(\pi)$ equals the Jacobian coordinates of $\phi + \pi$ where

$$\boldsymbol{A}(\phi) = \begin{pmatrix} 6 & x_1^2 & -2y_1 & x_1 & 0 & 0 \\ 9y_1 & 0 & -(y_1^2 + 9) & 3x_1y_1 & y_1 & -3x_1^2 \\ -x_1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

The alternative form $\boldsymbol{B}(\phi) \times \overline{\boldsymbol{u}}^T(\pi)$ follows immediately from $\boldsymbol{u}^T(\pi) = \boldsymbol{G} \times \overline{\boldsymbol{u}}^T(\pi)$ by setting $\boldsymbol{B}(\phi) = \boldsymbol{A}(\phi) \times \boldsymbol{G}$. This finishes the proof.

□

Lemma 4. *Let $\phi, \pi \in E(\mathbb{F}_p)$ be two points in affine coordinates such that $x(\phi) \neq x(\pi)$. Then the Jacobian coordinates of $\phi + \delta\pi$ where $\delta \in \{0,1\}$,*[11] *is given by $\boldsymbol{C}(\delta, \phi) \times \boldsymbol{u}^T(\pi)$ where $\boldsymbol{C}(\delta, \phi) \in \mathbb{F}_p^{3\times 6}$ is a matrix that depends only on δ, ϕ, defined by:*

$$\boldsymbol{C}(\delta, \phi) = \begin{pmatrix} 6\delta + (1-\delta)x(\phi) & \delta x^2(\phi) & -2\delta y(\phi) & \delta x(\phi) & 0 & 0 \\ 9\delta y(\phi) + (1-\delta)y(\phi) & 0 & -\delta(y(\phi)^2 + 9) & 3\delta x(\phi)y(\phi) & \delta y(\phi) & -3\delta x^2(\phi) \\ -\delta x(\phi) + (1-\delta) & \delta & 0 & 0 & 0 & 0 \end{pmatrix}$$

Equivalently, Jacobian coordinates of $\phi + \delta\pi$ are also given by $\boldsymbol{D}(\delta, \phi) \times \overline{\boldsymbol{u}}^T(\pi)$ where $\boldsymbol{D}(\delta, \phi) \in \mathbb{F}_p^{3\times(1+5n)}$ is defined as $\boldsymbol{D}(\delta, \phi) = \boldsymbol{C}(\delta, \phi) \times \boldsymbol{G}$.

Proof. The proof is analogous to the proof of Lem. 3. The only difference is that we need to consider the case where $\delta = 1$ and $\delta = 0$ separately. $\square$

Lemma 5. *Given a point $\pi \in E(\mathbb{F}_p)$ in Jacobian coordinates (X, Y, Z) (which is not unique) we can obtain a uniform representation of the same point as $\mathsf{diag}(\lambda^2, \lambda^3, \lambda) \times (X, Y, Z)^T$ for λ sampled uniformly from $\mathbb{F}_p^*$ where $\mathsf{diag}(\lambda^2, \lambda^3, \lambda)$ is a diagonal matrix with the entries $\lambda^2, \lambda^3, \lambda$ on the diagonal. More precisely,*

$$\mathsf{diag}(\lambda) = \begin{pmatrix} \lambda^2 & 0 & 0 \\ 0 & \lambda^3 & 0 \\ 0 & 0 & \lambda \end{pmatrix}$$

The proof follows from the definitions of Jacobian representation and we skip the details.

5.3 Decomposable Randomized Encodings: Definitions and Preliminaries

Randomized encodings allow us to represent a function f by a simpler function $\hat{f}$ such that the encoding $\hat{f}(x; \rho)$ (where ρ is some randomness independent of x) reveals $f(x)$ but nothing else about x beyond what is revealed by $f(x)$ itself. This section presents the formal framework for decomposable randomized encodings, following the definitions (some of them verbatim) from Ishai [Ish13].

Definition 13 (Randomized encoding (Definition 3.1, [Ish13]))**.** *Let $X, Y, \hat{Y}, R$ be finite sets and let $f : X \to Y$. A function $\hat{f} : X \times R \to \hat{Y}$ is a* randomized encoding *of f if it satisfies:*

- ***δ-Correctness.*** *There exists a decoder (possibly randomized)* $\mathsf{Dec} : \hat{Y} \to Y$ *such that for all $x \in X$ and $\rho \in R$,*

  $$\Pr[\mathsf{Dec}(\hat{f}(x, \rho)) \neq f(x)] \leqslant \delta$$

  where randomness is over the choices of ρ and the decoder. We skip mentioning δ when it is a negligible function of the appropriate parameters.

- ***ϵ-privacy.*** *There exists a randomized simulator* $\mathsf{Sim} : Y \to \hat{Y}$ *such that for all $x \in X$,*

  $$\mathsf{Sim}(f(x)) \approx_\epsilon \hat{f}(x, \rho),$$

  where $\rho \leftarrow R$ is uniform and $\approx_\epsilon$ denotes distributions that are statistically close. Again, we skip mentioning ϵ when it is a negligible function of the relevant parameters.

Additionally, a randomized encoding is said to be *efficient* if its encoding and decoding complexities are polynomial in the size of the input. In the literature, various relaxations of the above definition are considered. For example, one may consider a randomized encoding that is only computationally or statistically private. However, we focus on the perfect privacy, as it suffices for our purposes.

Next we define the decomposability property and some useful properties of randomized encodings.

[11] $\delta\pi$ denotes the point π if $\delta = 1$ and $\mathcal{O}$ if $\delta = 0$.

Definition 14 (Decomposable randomized encoding (DRE) (Definition 4.1, [Ish13])). *For $f : X_1 \times \cdots \times X_n \to Y$, a decomposable randomized encoding of f is one that has the form*

$$\hat{f}((x_1, \dots, x_n), \rho) = (\hat{f}_1(x_1, \rho), \dots, \hat{f}_n(x_n, \rho))$$

for some functions $\hat{f}_i : X_i \times R \to \hat{Y}_i$.

The decomposability property is what makes DRE particularly useful: each input component x_i can be encoded independently (though the encodings may share randomness ρ), and the evaluation algorithm can reconstruct $f(x_1, \dots, x_n)$ from these individual encodings. This structure is essential for the efficient garbled circuit constructions we present in the following sections.

Lemma 6 (Concatenation (Lemma 3.3, [Ish13])). *Suppose $\hat{f}_i(x, \rho_i)$ is a randomized encoding of $f_i(x)$ for $i = 1, \dots, k$. Then the function $\hat{f}(x, (\rho_1, \dots, \rho_k)) \overset{def}{=} (\hat{f}_1(x, \rho_1), \dots, \hat{f}_k(x, \rho_k))$ is a randomized encoding of $f(x) \overset{def}{=} (f_1(x), \dots, f_k(x))$.*

Lemma 7 (Composition (Lemma 3.4, [Ish13])). *Suppose $\hat{f}(x, \rho)$ is a randomized encoding of $f(x)$ and $\hat{f}'((x, \rho), \rho')$ is a randomized encoding of $\hat{f}(x, \rho)$ (viewing the latter as a deterministic function of (x, ρ)). Then $\hat{f}''(x, (\rho, \rho')) \overset{def}{=} \hat{f}'((x, \rho), \rho')$ is a randomized encoding of $f(x)$.*

5.4 Decomposable Randomized Encodings Constructions

Building on [FKN94], Ishai [Ish13] presents a simple decomposable encoding for summation in finite abelian groups. We state a slight generalization to *weighted summation* here.

Lemma 8 (DRE for weighted group summation (Generalization of Claim 4.2, [Ish13])). *Let $\mathcal{G}$ be a finite abelian group and let $f_{a_1,\dots,a_n} : \mathcal{G}^n \to \mathcal{G}$ be the group summation function $f_{a_1,\dots,a_n}(x_1, \dots, x_n) = \sum_{i=1}^{n} a_i x_i$.*[12] *Let $R = \{(\rho_1, \dots, \rho_n) \in \mathcal{G}^n : \sum_{i=1}^{n} a_i \rho_i = 0\}$. Then the function $\hat{f}_{a_1,\dots,a_n} : \mathcal{G}^n \times R \to \mathcal{G}^n$ defined by $\hat{f}_{a_1,\dots,a_n}((x_1, \dots, x_n), (\rho_1, \dots, \rho_n)) = (x_1 + \rho_1, \dots, x_n + \rho_n)$ is a decomposable encoding of $f_{a_1,\dots,a_n}$.*

Proof. As in Ishai [Ish13], it is easy to verify that $\hat{f}_{a_1,\dots,a_n}$ maps an input $x = (x_1, \dots, x_n)$ to a uniformly random input $x' \in \mathcal{G}^n$ such that $f_{a_1,\dots,a_n}(x') = f_{a_1,\dots,a_n}(x)$. Thus, we can let $\mathsf{Dec} = f_{a_1,\dots,a_n}$ and let $\mathsf{Sim}(y)$ output a random n-tuple in $f^{-1}_{a_1,\dots,a_n}(y)$. □

Remark 4. *We will actually use the above lemma on $f_{a_1,\dots,a_n}(x_1, \dots, x_n) = \sum_{i=1}^{n} a_i x_i$ with inputs $x_1 \dots x_n$ in Jacobian representation which is not unique for every group element. Thus, we randomize the representations in the DRE. More specifically, $\hat{f}_{a_1,\dots,a_n} : \mathbb{G}^n \times \mathbb{G}^n \times \mathbb{F}_p^{*n} \to \mathbb{G}^n$ defined by*

$$\hat{f}_{a_1,\dots,a_n}((x_1, \dots, x_n), (\rho_1, \dots, \rho_n), (\lambda_1, \dots, \lambda_n)) = (\mathsf{diag}(\lambda_1^2, \lambda_1^3, \lambda_1) \times (x_1 + \rho_1)^T, \dots, \mathsf{diag}(\lambda_n^2, \lambda_n^3, \lambda_n) \times (x_n + \rho_n)^T)$$

where $\mathsf{diag}(\lambda_i^2, \lambda_i^3, \lambda_i)$ is the diagonal matrix with $\lambda_i^2, \lambda_i^3, \lambda_i$ on the diagonal as defined in Lem. 5.

We will also need a DRE for private weights. However, we limit the inputs to be either 0 or 1 as it suffices for our purposes and keeps the DRE encoding function efficient.

Lemma 9 (DRE for private affine functions). *Let $\mathcal{G}$ be a finite abelian group and let $f : \{\mathcal{G} \times \{0, 1\}\}^n \to \mathcal{G}$ be the group summation function $f((a_1, x_1), \dots, (a_n, x_n)) = \sum_{i=1}^{n} x_i a_i$.*[13] *Let $R = \{(\rho_1, \dots, \rho_n) \in \mathcal{G}^n : \sum_{i=1}^{n} \rho_i = 0\}$. Then the function $\hat{f} : \{\mathcal{G} \times \{0, 1\}\}^n \times R \to \mathcal{G}^n$ defined by $\hat{f}(((a_1, x_1), \dots, (a_n, x_n)), (\rho_1, \dots, \rho_n)) = (x_1 a_1 + \rho_1, \dots, a_n x_n + \rho_n)$ is a decomposable encoding of f.*

[12]Where $+$ is the group operation and $a\rho = \overbrace{\rho + \ldots + \rho}^{a \text{ times}}$.

[13]Where $+$ is the group operation and $ax = 0$ if $x = 0$ and a otherwise.

Proof. The proof of this claim is identical to the proof of Lem. 8. This is a DRE with respect to the input blocks (a_i, x_i). It's decomposable because the i-th output depends only on (a_i, x_i) and ρ_i. $\square$

5.5 DRE for the Scalar Multiplication

For the BN254 group, we define the function h in Def. 15, and construct a DRE for this function.

Definition 15. *Let $h : (\mathbb{F}_q \times \{0,1\})^\ell \to \mathbb{G}$ be the function defined by (with n, $\ell = 5n + 1$ as in Sec. 5.2):*[14]

$$h((r, \overline{u}_1(\pi)), (r, \overline{u}_2(\pi)) \cdots (r, \overline{u}_\ell(\pi))) \longmapsto r\pi.$$

The function h can be expressed as a matrix multiplication with a transition from field to group operations. The computation proceeds in two stages:

1. ***Field operations:** For each $i \in \{0, \dots, n-1\}$, compute the three Jacobian coordinates of $r_i\pi + \rho_i$ using matrix multiplication over $\mathbb{F}_p$:*

$$\mathsf{diag}(\lambda_i) \times \boldsymbol{D}(r_i, \rho_i) \times \overline{\boldsymbol{u}}^T(\pi)$$

where $\boldsymbol{N}(r_i) \in \mathbb{F}_p^{3\times\ell}$ is the matrix defined in Lem. 2 and $r_0 \cdots r_{n-1}$ is the bit decomposition of r; i.e. $\sum_i 2^i \cdot r_i = r$.

2. ***Group operations:** Apply weighted group sum using as the group operation:*

$$\sum_{i=0}^{n-1} 2^i (r_i\pi + \rho_i) = r\pi$$

where each $r_i\pi$ is represented by its Jacobian coordinates (X, Y, Z) computed in Step 1 and the operation $\sum_{i=0}^{n-1} 2^i(\cdot)$ uses group operations (elliptic curve addition and scalar multiplication in the group $\mathbb{G}$).

Informally, for each i, the computation $\boldsymbol{N}(r_i) \times \overline{\boldsymbol{u}}^T(\pi)$ yields the three field elements $(X, Y, Z) \in \mathbb{F}_p^3$ representing the Jacobian coordinates of the group element $r_i\pi$. The final weighted sum $\sum_{i=0}^{n-1} 2^i(r_i\pi)$ combines these group elements using elliptic curve addition, which operates on all three coordinates simultaneously.

Theorem 4. *There exists a function $\hat{h}$ such that $\hat{h} : (\mathbb{F}_q \times \{0,1\})^\ell \times (R, \Lambda, S) \to \mathbb{F}_p^{3n\ell}$ is a DRE for h as defined in Def. 15 where R, Λ, S is appropriately sampled randomness used in the encoding.*

Proof. We start by describing the function $\hat{h}$ in more detail. More specifically, we define the function

$$\hat{h}\left((r, \overline{u}_1(\pi)), (r, \overline{u}_2(\pi)) \cdots (r, \overline{u}_\ell(\pi)), R, \Lambda, S\right) = \left(\hat{h}_1(r, \overline{u}_1(\pi), R, \Lambda, S), \dots, \hat{h}_\ell(r, \overline{u}_\ell(\pi), R, \Lambda, S)\right)$$

where $\hat{h}_k(r, \overline{u}_k(\pi), R, \Lambda, S) = \{D_{i,j,k} \cdot \overline{u}_k(\pi) + s_{i,j,k}\}_{i,j}$, where $\boldsymbol{D}_i = \mathsf{diag}(\lambda_i^3, \lambda_i^2, \lambda_i) \times \boldsymbol{D}(r_i, \rho_i)$ and $i \in \{0, \dots, n-1\}$, $j \in \{1, 2, 3\}$, $k \in \{1, \dots, 5n+1\}$ and $D_{i,j,k}$ is the $(j,k)^{th}$ entry in $\boldsymbol{D}_i$.

Here we set the randomness such that $R = \{(\rho_1, \dots, \rho_n) \in (\mathbb{G} \setminus \{\mathcal{O}\})^n : \sum_{i=1}^n \rho_i = 0\}$, $\Lambda = (\lambda_1, \dots, \lambda_n) \in (\mathbb{F}_p \setminus \{0\})^n$ and $S = \{(s_{i,j,k}\}_{i=1,j=1,k=1}^{n,3,\ell} \in \mathbb{F}_p^{3n\ell} : \forall i, j \sum_{k=1}^{\ell} s_{i,j,k} = 0\}$.

[14]The inputs of h are actually from $\mathbb{F}_q \times \{0,1\}^\ell$ but we repeat the input r to highlight the decomposability.

Decoding Function. The decoding function $\mathsf{Dec} : \mathbb{F}_p^{3n\ell} \to \mathbb{G}$ on input $\gamma_1 \dots \gamma_\ell$ where $\gamma_k = \{t_{i,j,k}\}_{i,j}$ is either $\hat{h}_k(r, 0, R, \Lambda, S)$ or $\hat{h}_k(r, 1, R, \Lambda, S)$ is defined as follows:

1. For each $i \in \{0, \dots, n-1\}$ and $j \in \{1, 2, 3\}$, compute the j-th coordinate of the randomized Jacobian representation:
$$\hat{t}_{i,j} = \sum_{k=1}^{\ell} t_{i,j,k}$$
This correctly recovers the j-th coordinate since $\sum_{k=1}^{\ell} s_{i,j,k} = 0$ by the constraint on S.
2. For each $i \in \{0, \dots, n-1\}$, form the point $Q_i = (\hat{t}_{i,1}, \hat{t}_{i,2}, \hat{t}_{i,3}) \in \mathbb{F}_p^3$, which represents the group element $\mathsf{diag}(\lambda_i^3, \lambda_i^2, \lambda_i) \times (r_i\pi + \rho_i)$.
3. Apply the weighted group sum to recover $r\pi$:
$$\sum_{i=0}^{n-1} 2^i Q_i = r\pi$$
This correctly computes $r\pi$ since $\sum_{i=0}^{n-1} 2^i \rho_i = 0$ by the constraint on R.

Next we derive this DRE for h using Lem. 8 and Lem. 9.

Step 1: DRE for weighted group sum using Lem. 8. Let $r = \sum_{i=0}^{n-1} r_i 2^i \in \mathbb{F}_q$ be the binary decomposition of r. We want to compute $r\pi = \sum_{i=0}^{n-1} 2^i(r_i\pi)$. Define the function $f : \mathbb{G}^n \to \mathbb{G}$ by:
$$f(r_0\pi, \dots, r_{n-1}\pi) = \sum_{i=0}^{n-1} 2^i(r_i\pi) = r\pi$$
with weights $a_i = 2^i$ for $i \in \{0, \dots, n-1\}$. By Lem. 8 and Remark 4, there exists a DRE $\hat{f} : \mathbb{G}^n \times R_1 \to \mathbb{G}^n$ where $R_1 = \{(\rho_0, \dots, \rho_{n-1}) \in \mathbb{G}^n, (\lambda_0, \dots, \lambda_{n-1}) \in \mathbb{F}_p^{*n} : \sum_{i=0}^{n-1} 2^i \rho_i = 0\}$, where
$$\hat{f}((r_0\pi, \dots, r_{n-1}\pi), R_1 = ((\rho_0, \dots, \rho_{n-1}), (\lambda_0, \dots, \lambda_{n-1}))) = (\hat{f}_0(r_0\pi, R_1), \dots, \hat{f}_{n-1}(r_{n-1}\pi, R_1)$$
and for each i, $\hat{f}_i(r_i\pi, R_1) = \mathsf{diag}(\lambda_i^3, \lambda_i^2, \lambda_i) \times (r_i\pi + \rho_i)^T$. The decoding function is $\mathsf{Dec}_1 : \mathbb{G}^n \to \mathbb{G}$ defined by $\mathsf{Dec}_1((y_0, \dots, y_{n-1})) = \sum_{i=0}^{n-1} 2^i y_i$, which correctly computes $r\pi$ since $\sum_{i=0}^{n-1} 2^i \rho_i = 0$.

Step 2: DRE for $\mathsf{diag}(\lambda_i^3, \lambda_i^2, \lambda_i) \times (r_i\pi + \rho_i)^T$ using Lem. 9. For each $i \in \{0, \dots, n-1\}$, we need to encode the group element $r_i\pi + \rho_i$. Since $r_i \in \{0, 1\}$, we have:
$$r_i\pi + \rho_i = \begin{cases} \pi + \rho_i & \text{if } r_i = 1 \\ \rho_i & \text{if } r_i = 0 \end{cases}$$
For each i, let E_i be the event $\rho_i = \pi$. If $\neg E_i$ then we can apply Lem. 4 to express the randomized Jacobian coordinates of $\mathsf{diag}(\lambda_i^3, \lambda_i^2, \lambda_i) \times (r_i\pi + \rho_i)^T$ (setting with $\phi = \rho_i$ and $\delta = r_i$ in Lem. 4) as a linear function of $\boldsymbol{u}(\pi)$. Specifically:
$$\mathsf{diag}(\lambda_i^3, \lambda_i^2, \lambda_i) \times \boldsymbol{D}(r_i, \rho_i) \times \overline{\boldsymbol{u}}^T(\pi)$$
which can be written as $\boldsymbol{D}_i \times \overline{\boldsymbol{u}}^T(\pi)$ where $\boldsymbol{D}_i \overset{def}{=} \mathsf{diag}(\lambda_i^3, \lambda_i^2, \lambda_i) \times \boldsymbol{D}(r_i, \rho_i) \in \mathbb{F}_p^{3\times(1+5n)}$.

For each coordinate $j \in \{1, 2, 3\}$ (corresponding to X, Y, Z), let $\boldsymbol{D}_{i,j}$ be the j-th row of $\boldsymbol{D}_i$. Then, the j-th coordinate of the randomized Jacobian coordinates is:

$$\boldsymbol{D}_{i,j} \times \overline{\boldsymbol{u}}^T(\pi) = \sum_{k=1}^{1+5n} D_{i,j,k} \cdot \overline{u}_k(\pi)$$

where $D_{i,j,k}$ is the k-th entry of the row vector $\boldsymbol{D}_{i,j}$, and $\overline{u}_k(\pi)$ is the k-th component of $\overline{\boldsymbol{u}}(\pi)$. Now, for each $i \in \{0, \ldots, n-1\}$ and $j \in \{1, 2, 3\}$, we apply Lem. 9 to encode the function:

$$g_{i,j} : (\mathbb{F}_p \times \{0, 1\})^{1+5n} \to \mathbb{F}_p$$

defined by $g_{i,j}((D_{i,j,1}, \overline{u}_1(\pi)), \ldots, (D_{i,j,1+5n}, \overline{u}_{1+5n}(\pi))) = \sum_{k=1}^{1+5n} D_{i,j,k} \cdot \overline{u}_k(\pi)$. Note that $D_{i,j,k}$ are known values that depend on r_i and the randomness ρ_i and λ_i from Step 1. The function $g_{i,j}$ takes as input pairs $(D_{i,j,k}, \overline{u}_k(\pi))$ where $D_{i,j,k} \in \mathbb{F}_p$ is a field element (the coefficient) and $\overline{u}_k(\pi) \in \{0, 1\}$ is a bit. By Lem. 9, there exists a DRE $\hat{g}_{i,j} : ((\mathbb{F}_p \times \{0, 1\}))^{1+5n} \times S_{i,j} \to \mathbb{F}_p^{1+5n}$ where $S_{i,j} = \{(s_{i,j,1}, \ldots, s_{i,j,1+5n}) \in \mathbb{F}_p^{1+5n} : \sum_{k=1}^{1+5n} s_{i,j,k} = 0\}$, defined by:

$$\begin{aligned}\hat{g}_{i,j}((D_{i,j,1}, \overline{u}_1(\pi)), \ldots, (D_{i,j,1+5n}, \overline{u}_{1+5n}(\pi))), S_{i,j} = (s_{i,j,1}, \ldots, s_{i,j,1+5n})) = \\ (D_{i,j,1} \cdot \overline{u}_1(\pi) + s_{i,j,1}, \ldots, D_{i,j,1+5n} \cdot \overline{u}_{1+5n}(\pi) + s_{i,j,1+5n})\end{aligned}$$

The decoding function is $\mathsf{Dec}_{i,j} : \mathbb{F}_p^{1+5n} \to \mathbb{F}_p$ defined by $\mathsf{Dec}_{i,j}((y_1, \ldots, y_{1+5n})) = \sum_{k=1}^{1+5n} y_k$, which, conditioned on the fact that Lem. 4 can be applied (i.e. $\neg E_i$), correctly computes the correct j-th coordinate since $\sum_{k=1}^{1+5n} s_{i,j,k} = 0$.

Composition and final DRE. By Lem. 7 and Lem. 6, we can compose the DREs from Step 1 and Step 2 to obtain the final DRE $\hat{h}$ for h.

Correctness and Security. Our analysis holds conditioned on $\neg E_i$ for each $i = 1, \ldots, n$. $\Pr[E_i] = \Pr[\rho_i = \pi] = \frac{1}{|\mathbb{G}|-1}$, thus the correctness and privacy errors of the final DRE are $\delta = \epsilon = \frac{n}{|\mathbb{G}|-1}$. □

5.6 Completing the Garbled Circuit

We wish to design an efficient Garbled Circuit that on public input an (on-chain) Lamport signature on $\pi = (x(\pi), y(\pi))$ outputs $r\pi$. Here $r \in \mathbb{F}_q^*$ is a private input known at the time garbling that is considered hard-coded in the circuit.

Construction 2. *The garbled circuit* GC *for the function* $f_r : \mathbb{G} \to \mathbb{G}$, $f_r(\pi) = r\pi$ *will be as follows:*

$\mathsf{Garble}(r) \to \mathsf{ct}_{GC}, \mathsf{ek}$*: Outputs a Lamport secret key*

$$\mathsf{ek} = \begin{pmatrix} L^0_{x,0} & \ldots & L^0_{x,n-1} & L^0_{y,0} & \ldots & L^0_{y,n-1} \\ L^1_{x,0} & \ldots & L^1_{x,n-1} & L^1_{y,0} & \ldots & L^1_{y,n-1} \end{pmatrix}$$

and the garbled circuit $\mathsf{ct}_{\mathsf{GC}}$ *computed in three steps:*

1. ***Elliptic curve validation:*** *The garbled circuit first verifies that the input point* $\pi = (x(\pi), y(\pi))$ *lies on the curve* E*, i.e., it checks that* $y(\pi)^2 \equiv x(\pi)^3 + 3 \pmod{p}$.
2. ***Binary decomposition:*** *If the validation passes, the circuit computes the binary decomposition* $\overline{\boldsymbol{u}}(\pi) \in \{0, 1\}^{1+5n}$ *of* $\boldsymbol{u}(\pi) = (1, x(\pi), y(\pi), x(\pi)^2, y(\pi)^2, x(\pi)y(\pi))$*. If not, the circuit outputs the binary decomposition of* $\boldsymbol{u}(g) = (1, x(g), y(g), x(g)^2, y(g)^2, x(g)y(g))$ *where* $g \in \mathbb{G}_1$ *is some fixed point in the curve. These two steps are implemented using a privacy-free Boolean garbled circuit. Evaluating this circuit outputs the labels* $L^{\overline{u}_k(\pi)}_{\overline{u},k}$ *for* $k \in [\ell]$.

3. ***DRE encoding:*** *For each $k \in \{1, \dots, \ell\}$ where $\ell = 1 + 5n$, the garbled circuit comes hardwired with the DRE encoding from Thm. 4. Specifically:*
   - *For each bit position $k \in [\ell]$ where $\ell = 1 + 5n$, the garbler precomputes $\hat{h}_k((r, 0), \omega)$ and $\hat{h}_k((r, 1), \omega)$.*
   - *The garbled circuit contains an encryption of $\hat{h}_k((r, 0), \omega)$ under the 0-label $L^0_{\overline{u},k}$ for $\overline{u}_k(\pi)$.*
   - *The garbled circuit contains an encryption of $\hat{h}_k((r, 1), \omega)$ under the 1-label $L^1_{\overline{u},k}$ for $\overline{u}_k(\pi)$.*

$\mathsf{Encode}(\mathsf{ek}, \pi) \rightarrow L$: *Outputs the Lamport signature of π*

$$L = \left(L^{x_0}_{x,0} \quad \dots \quad L^{x_{n-1}}_{x,n-1} \quad L^{y_0}_{y,0} \quad \dots \quad L^{y_{n-1}}_{y,n-1}\right)$$

$\mathsf{Eval}(\mathsf{ct}_{GC}, L_\pi) \rightarrow r\pi$: *The evaluator, given the input labels L_π, evaluates the Boolean circuits to obtain labels for $\overline{u}(\pi)$, and decrypts the appropriate encryptions to obtain $\hat{h}_k(r, \overline{u}_k(\pi))$ for each k. The evaluator then decodes the DRE as follows (Thm. 4):*

- *We recast $h_k(r, \overline{u}_k(\pi), \omega)$ as $\{t_{i,j,k}\}_{i \in [n], j \in \{1,2,3\}}$. Then the Prover aggregates shares to cancel the $s_{i,j,k}$-terms:*

$$\widehat{t}_{i,j} = \sum_{k=1}^{\ell} t_{i,j,k}.$$

- *Interpreting $(\widehat{t}_{i,1}, \widehat{t}_{i,2}, \widehat{t}_{i,3})$ as the Jacobian coordinates of a masked point Q_i, the final output is obtained by the usual bit-weighted recombination*

$$\mathrm{out} = \sum_i 2^i Q_i.$$

Since $Q_i = r_i \pi + \rho_i$ and $\sum_i 2^i \rho_i = 0$, then $\mathrm{out} = r\pi$.

Intuitively, the heavy work (group operations and most algebra) is pushed to the Prover and to the Verifier's offline preprocessing, while the online garbled circuit mainly performs validation, low-level arithmetic for feature extraction, and symmetric-key decryptions for table selection to output the precomputed DRE components.

Theorem 5. *Constr. 2 is an adaptively secure garbling scheme (Def. 3) for the function $f_r : \mathbb{G} \rightarrow \mathbb{G}$, $f_r(\pi) = r\pi$ in the random oracle model.*

The proof follows directly from Thm. 4 and the security of Yao's garbling scheme [LP09].[15] For adaptive security we rely on standard random oracle techniques [BHR12a] (namely equivocal encryption).

Remark 5 (Optimizations for the Randomized Encoding)**.** *In practice we integrate the following optimizations:*

- *Several entries in the matrix $\boldsymbol{D}$ are 0. In our implementation the inner products only need to grow with the number of nonzero entries. This reduces garbled circuit size by a factor of 1.87.*
- *Rather than giving two ciphertexts encrypting $\hat{h}_i((r, 0), R, S)$ and $\hat{h}_i((r, 1), R, S)$ we just encrypt one of them and let the appropriate shifts be the output of a Pseudorandom Function (e.g. a random oracle) on the corresponding label, such that the latter can be computed locally. This reduces garbled circuit size by a factor of 2.*

[15] For the optimization we propose, i.e. instead of letting the circuit compute the $\hat{h}_k((r, \overline{u}_k), \omega)$'s we precompute them and encrypt them under the corresponding labels, the simulator may just encrypt the simulated $\hat{h}_k$'s for the actual $\overline{u}_k$'s and provide encryption of 0 for the other bit.

Efficiency. We provide a theoretical estimation of our garbled circuit size. Since it is hard to theoretically estimate the size of the circuit for (1) and (2) of the construction (Elliptic Curve validation and Binary Decomposition) we postpone this for the experimental evaluation in Sec. 9. However, the main cost stems from (3), the DRE Encoding. Taking into account our optimizations, the latter consists of:

- For each $i = 0, \dots, n-1$
    - $3n + 1$ $\log |\mathbb{F}|$-sized ciphertexts for the X coordinate.
    - $4n + 1$ $\log |\mathbb{F}|$-sized ciphertexts for the Y coordinate.
    - $n + 1$ $\log |\mathbb{F}|$-sized ciphertexts for the Z coordinate.

Here $\mathbb{F}$ is the field generated by the $E/\mathbb{F}_p$ group, therefore $|\mathbb{F}| = 254$ and $n = 254$. This gives us:

$$254 \cdot (8 \cdot 254 + 3) \cdot 254 \text{ bits} = 15.65 \text{ MiB}$$

This is validated empirically in Sec. 9.

6 BABE Protocol

6.1 Honest Setup Protocol

A BitVM-core protocol (Sec. 3) has two phases: a setup phase and a proving phase. The setup phase consists of off-chain interaction between the Prover and the Verifier and terminates with some Bitcoin being locked (Fig. 4). The proving phase consists of on-chain interaction between the Prover and the Verifier and ends with the Prover posting a transaction to withdraw the locked Bitcoin. To explain the core aspects of the protocol, we first describe the protocol assuming that the setup is run honestly by both Prover and Verifier (Algs. 1 and 2), but either party may be malicious in the proving phase. In Sec. 6.2, we discuss how the Prover and Verifier can verify that the setup was run correctly and abort if it was not.

6.1.1 Setup Phase

The setup phase protocol is described in Alg. 1. The Prover initiates the setup by sending his public key pk_P to the Verifier. The Verifier creates the application-specific statement $\boldsymbol{x}$ to be proven. For example, in the lending application from Sec. 1, when the borrower is the Prover, the statement is that the borrower repaid his loan on Ethereum. The Verifier samples a secret msg and creates the witness encryption ciphertext ($\mathsf{ct}_{\mathsf{setup}}$ in Constr. 1) that doesn't depend on the proof. He also creates the garbled circuit (Constr. 2) to compute $\mathsf{ct}_{\mathsf{prove}}$, the remaining part of the ciphertext, generating the encoding key ek and the garbled circuit ciphertext $\mathsf{ct}_{\mathsf{GC}}$. The Verifier hashes the secret msg and the garbled circuit's encoding key ek to be used in the hashlock scripts. Then, both parties create a set of transaction skeletons[16] $\mathcal{T}$ shown in Fig. 5. The Prover pre-signs the transaction skeletons $\overline{\mathsf{tx}}_{\mathsf{ChallengeAssert}}$ and $\overline{\mathsf{tx}}_{\mathsf{NoWithdraw}}$ (ones that the Verifier may post during the proving phase). Similarly, the Verifier pre-signs the transaction skeletons $\overline{\mathsf{tx}}_{\mathsf{Assert}}$ and $\overline{\mathsf{tx}}_{\mathsf{Withdraw}}$ (ones that the Prover may post during the proving phase). At the end, the Prover stores the state st_P consisting of his secret keys, the pre-signatures sent by the Verifier, the witness encryption ciphertext $\mathsf{ct}_{\mathsf{setup}}$, and the garbled circuit ciphertext $\mathsf{ct}_{\mathsf{GC}}$. The Verifier's state st_V consists of his secret keys, and the pre-signatures sent by the Prover.

[16]Recall: transaction skeleton is transaction without transaction witness.

Algorithm 1 Setup algorithms (honest Prover and Verifier)

```
1:  function Gen($\mathcal{R}$)
2:      return Groth16.Gen($\mathcal{R}$)
3:  end function

4:  procedure $P_{\mathsf{Setup}}(\mathsf{crs})$                                         ▷ Run by Prover
5:      $(\mathsf{sk}_P, \mathsf{pk}_P) \leftarrow \mathsf{Sig}_{BTC}.\mathtt{Gen}(1^\lambda)$                ▷ Sample signing key
6:      send $(\mathsf{pk}_P)$ to Verifier

        Upon receiving $(\mathsf{pk}_V, h_{\mathsf{msg}}, \mathsf{epk}, \mathsf{ct}_{\mathsf{setup}}, \mathsf{ct}_{\mathsf{GC}})$ from Verifier:
7:      $(\mathsf{lsk}_P, \mathsf{lpk}_P) \leftarrow \mathsf{LampSig.Gen}(1^\lambda)$                    ▷ Sample Lamport key
8:      $(\mathcal{T}, \mathcal{S}) \leftarrow \mathsf{CreateTxSet}(\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{lpk}_P, h_{\mathsf{msg}}, \mathsf{epk})$    ▷ See Alg. 3
9:      $\mathsf{presigs}_P \leftarrow \mathsf{SignTxs}_P(\mathsf{sk}_P, \mathcal{T})$                       ▷ See Alg. 3
10:     send $(\mathsf{pk}_P, \mathsf{lpk}_P, \mathsf{presigs}_P)$ to Verifier

        Upon receiving $(\mathsf{presigs}_V)$ from Verifier:
11:     Sign $\mathsf{tx}_{\mathsf{Deposit}}$ and submit to Bitcoin via $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{Deposit}})$
12:     $\mathsf{st}_P \leftarrow (\mathsf{sk}_P, \mathsf{lsk}_P, \mathsf{presigs}_V, \mathsf{ct}_{\mathsf{setup}}, \mathsf{ct}_{\mathsf{GC}})$
13:     return $(\boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P)$
14: end procedure

15: procedure $V_{\mathsf{Setup}}(\mathsf{crs})$                                         ▷ Run by Verifier
        Upon receiving $(\mathsf{pk}_P)$ from Prover:
16:     $\boldsymbol{x} \leftarrow \mathsf{GenStmt}(\mathsf{pk}_P)$          ▷ Application-specific: map Prover to statement
17:     $(\mathsf{sk}_V, \mathsf{pk}_V) \leftarrow \mathsf{Sig}_{BTC}.\mathtt{Gen}(1^\lambda)$                ▷ Sample signing key
18:     $\mathsf{msg} \leftarrow\$ \mathcal{M}, r \leftarrow\$ \mathbb{F}_q^*$                               ▷ Sample secrets
19:     $\mathsf{ct}_{\mathsf{setup}} \leftarrow \mathsf{WE.Enc}_{\mathsf{setup}}(\mathsf{crs}, \boldsymbol{x}, \mathsf{msg}, r)$               ▷ WE ciphertext (Constr. 1)
20:     $\mathsf{ct}_{\mathsf{GC}}, \mathsf{ek} \leftarrow \mathsf{Garble}(r)$      ▷ Garbled circuit ciphertext and encoding key (Sec. 5.6)
21:     $h_{\mathsf{msg}} \leftarrow \mathsf{Hash}_{BTC}(\mathsf{msg})$                          ▷ Hash message for hashlock
22:     for $j \in \{1, \ldots, 2n\}, b \in \{0, 1\}$ do
23:         $\mathsf{epk}_j^b \leftarrow \mathsf{Hash}_{BTC}(\mathsf{ek}_j^b)$                    ▷ Hash input labels for hashlock
24:     end for
25:     send $(\mathsf{pk}_V, h_{\mathsf{msg}}, \mathsf{epk}, \mathsf{ct}_{\mathsf{setup}}, \mathsf{ct}_{\mathsf{GC}})$ to Prover

        Upon receiving $(\mathsf{pk}_P, \mathsf{lpk}_P, \mathsf{presigs}_P)$ from Prover:
26:     $(\mathcal{T}, \mathcal{S}) \leftarrow \mathsf{CreateTxSet}(\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{lpk}_P, h_{\mathsf{msg}}, \mathsf{epk})$
27:     $\mathsf{presigs}_V \leftarrow \mathsf{SignTxs}_V(\mathsf{sk}_V, \mathcal{T})$
28:     $\mathsf{st}_V \leftarrow (\mathsf{sk}_V, \mathsf{ek}, \mathsf{presigs}_P)$
29:     send $(\mathsf{presigs}_V)$ to Prover
30:     return $(\boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_V)$
31: end procedure
```

6.1.2 Proving Phase

The proving phase protocol is described in Alg. 2. In the proving phase, the Prover generates a Groth16 proof (π_1, π_2, π_3) using the witness for the statement $\boldsymbol{x}$ that was agreed upon during the setup phase. The proving phase involves the following transactions (shown in Fig. 5) posted on Bitcoin:

1. Assert: used by the Prover to post the proof element π_1.
2. ChallengeAssert: used by the Verifier to post the input labels for the proof π_1. The Bitcoin script verifies that the input labels are for the same proof π_1 that the Prover posted.
3. If the Prover's proof is valid:
   (a) WronglyChallenged: The Prover evaluates the garbled circuit to compute $\mathsf{ct}_{\mathsf{prove}} = r\pi_1$, decrypts the secret msg (Constr. 1), then posts this transaction. The Bitcoin script requires the

Algorithm 2 Prove algorithms

```
1:  procedure $P_{\texttt{Prove}}(\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P, \boldsymbol{w})$    ▷ Run by Prover
2:      Parse $\mathsf{st}_P = (\mathsf{sk}_P, \mathsf{lsk}_P, \mathsf{presigs}_V, \mathsf{ct}_{\mathsf{setup}})$ and $\mathsf{presigs}_V = (\sigma^V_{\mathsf{Withdraw}})$
3:      $(\pi_1, \pi_2, \pi_3) \leftarrow \mathsf{Groth16}.\texttt{Prove}(\mathsf{crs}, \boldsymbol{x}, \boldsymbol{w})$
4:      $w_{\mathsf{Assert}} \leftarrow \mathsf{LampSig.Sign}(\mathsf{lsk}_P, \pi_1)$    ▷ Compute Lamport signature
5:      Post $\mathsf{tx}_{\mathsf{Assert}}$ with transaction witness $w_{\mathsf{Assert}}$: call $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{Assert}})$

        Upon seeing $\mathsf{tx}_{\mathsf{Assert}}$ and $\Delta_2$ new blocks after $\mathsf{tx}_{\mathsf{Assert}}$ in $\mathcal{L}_P$:
6:      $w_{\mathsf{Withdraw}} \leftarrow (\sigma^P_{\mathsf{Withdraw}}, \sigma^V_{\mathsf{Withdraw}})$ where $\sigma^P_{\mathsf{Withdraw}} \leftarrow \mathsf{Sig}_{BTC}.\mathsf{Sign}(\mathsf{sk}_P, \overline{\mathsf{tx}}_{\mathsf{Withdraw}})$
7:      Post $\mathsf{tx}_{\mathsf{Withdraw}}$ with transaction witness $w_{\mathsf{Withdraw}}$: call $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{Withdraw}})$
8:      return 1

        Upon seeing $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ in $\mathcal{L}_P$:
9:      Extract input labels $L$ from the transaction witness of $\mathsf{tx}_{\mathsf{ChallengeAssert}}$
10:     $\mathsf{ct}_{\mathsf{prove}} \leftarrow \mathsf{EvalGC}(\mathsf{ct}_{\mathsf{GC}}, L)$    ▷ Evaluate garbled circuit (Sec. 5.6)
11:     $\mathsf{msg} \leftarrow \mathsf{WE.Dec}(\mathsf{ct}_{\mathsf{setup}}, \mathsf{ct}_{\mathsf{prove}}, \pi_2, \pi_3)$    ▷ Decrypt message (Constr. 1)
12:     $w_{\mathsf{WronglyChallenged}} \leftarrow (\sigma^P_{\mathsf{WronglyChallenged}}, \mathsf{msg})$ where $\sigma^P_{\mathsf{WronglyChallenged}} \leftarrow \mathsf{Sig}_{BTC}.\mathsf{Sign}(\mathsf{sk}_P, \overline{\mathsf{tx}}_{\mathsf{WronglyChallenged}})$
13:     Post $\mathsf{tx}_{\mathsf{WronglyChallenged}}$ with transaction witness $w_{\mathsf{WronglyChallenged}}$: call $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{WronglyChallenged}})$
14: end procedure

15: procedure $V_{\texttt{Prove}}(\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_V)$    ▷ Run by Verifier
        Upon seeing $\mathsf{tx}_{\mathsf{Assert}}$ in $\mathcal{L}_V$:
16:     Parse $\mathsf{st}_V = (\mathsf{sk}_V, \mathsf{ek}, \mathsf{presigs}_P)$ and $\mathsf{presigs}_P = (\sigma^P_{\mathsf{ChallengeAssert}}, \sigma^P_{\mathsf{NoWithdraw}})$
17:     Extract $\pi_1$ and Lamport signature $\mu$ from the transaction witness of $\mathsf{tx}_{\mathsf{Assert}}$
18:     $L \leftarrow \mathsf{Encode}(\mathsf{ek}, \pi_1)$    ▷ Compute input labels (Sec. 5.6)
19:     $w_{\mathsf{ChallengeAssert}} \leftarrow (\sigma^P_{\mathsf{ChallengeAssert}}, \sigma^V_{\mathsf{ChallengeAssert}}, \mu, L)$ where $\sigma^V_{\mathsf{ChallengeAssert}} \leftarrow \mathsf{Sig}_{BTC}.\mathsf{Sign}(\mathsf{sk}_V, \overline{\mathsf{tx}}_{\mathsf{ChallengeAssert}})$
20:     Post $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ with transaction witness $w_{\mathsf{ChallengeAssert}}$: call $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{ChallengeAssert}})$

        Upon seeing $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ and $\Delta_1$ new blocks after $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ in $\mathcal{L}_V$:
21:     $w_{\mathsf{NoWithdraw}} \leftarrow (\sigma^P_{\mathsf{NoWithdraw}}, \sigma^V_{\mathsf{NoWithdraw}})$ where $\sigma^V_{\mathsf{NoWithdraw}} \leftarrow \mathsf{Sig}_{BTC}.\mathsf{Sign}(\mathsf{sk}_V, \overline{\mathsf{tx}}_{\mathsf{NoWithdraw}})$
22:     Post $\mathsf{tx}_{\mathsf{NoWithdraw}}$ with transaction witness $w_{\mathsf{NoWithdraw}}$: call $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{NoWithdraw}})$
23:     return 1
24: end procedure
```

Prover to provide the decrypted secret msg.

(b) Withdraw: The Prover posts this transaction to withdraw the locked Bitcoin. This transaction can only be posted Δ_2 blocks after he posted Assert. This timelock gives the Verifier enough time to "stop" a malicious Prover.

4. If the Prover's proof is invalid:

   (a) NoWithdraw: This prevents the Prover from ever posting the Withdraw transaction. This transaction can only be posted Δ_1 blocks after the Verifier posted ChallengeAssert, which gives the Prover enough time to post WronglyChallenged if he decrypted the secret msg.

6.1.3 Transaction Graph

The transactions posted during the proving phase form a graph where the outputs of one transaction are the inputs of another (this graph is shown in Fig. 5). A detailed specification of the transactions is given in App. A.1. This section describes the locking scripts used and how they ensure the protocol's security.

Deposit **transaction.** This locks v amount of Bitcoin. This transaction is posted at the end of the setup phase. The output of this transaction carries a locking script $\mathsf{CheckSig}(\mathsf{pk}_P) \wedge \mathsf{CheckSig}(\mathsf{pk}_V)$ which

Algorithm 3 Helper functions for Algs. 1 and 2

```
1: function CreateTxSet($\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{lpk}_P, h_{\mathsf{msg}}, h_{\mathsf{ek}}$)
2:     Construct transaction skeletons $\overline{\mathsf{tx}}_{\mathsf{Deposit}}, \overline{\mathsf{tx}}_{\mathsf{Assert}}, \overline{\mathsf{tx}}_{\mathsf{ChallengeAssert}}, \overline{\mathsf{tx}}_{\mathsf{NoWithdraw}}, \overline{\mathsf{tx}}_{\mathsf{WronglyChallenged}}, \overline{\mathsf{tx}}_{\mathsf{Withdraw}}$ as in App. A.1
3:     $\mathcal{T} := \{\overline{\mathsf{tx}}_{\mathsf{Deposit}}, \overline{\mathsf{tx}}_{\mathsf{Assert}}, \overline{\mathsf{tx}}_{\mathsf{ChallengeAssert}}, \overline{\mathsf{tx}}_{\mathsf{NoWithdraw}}, \overline{\mathsf{tx}}_{\mathsf{WronglyChallenged}}, \overline{\mathsf{tx}}_{\mathsf{Withdraw}}\}$
4:     $\mathcal{S} := \{\overline{\mathsf{tx}}_{\mathsf{Withdraw}}\}$
5:     return $(\mathcal{T}, \mathcal{S})$
6: end function

7: function SignTxs$_P$($\mathsf{sk}_P, \mathcal{T}$)
8:     $\sigma^P_{\mathsf{ChallengeAssert}} := \mathsf{Sig}_{BTC}.\mathsf{Sign}(\mathsf{sk}_P, \overline{\mathsf{tx}}_{\mathsf{ChallengeAssert}})$
9:     $\sigma^P_{\mathsf{NoWithdraw}} := \mathsf{Sig}_{BTC}.\mathsf{Sign}(\mathsf{sk}_P, \overline{\mathsf{tx}}_{\mathsf{NoWithdraw}})$
10:    return $(\sigma^P_{\mathsf{ChallengeAssert}}, \sigma^P_{\mathsf{NoWithdraw}})$
11: end function

12: function SignTxs$_V$($\mathsf{sk}_V, \mathcal{T}$)
13:    $\sigma^V_{\mathsf{Assert}} := \mathsf{Sig}_{BTC}.\mathsf{Sign}(\mathsf{sk}_V, \overline{\mathsf{tx}}_{\mathsf{Assert}})$
14:    $\sigma^V_{\mathsf{Withdraw}} := \mathsf{Sig}_{BTC}.\mathsf{Sign}(\mathsf{sk}_V, \overline{\mathsf{tx}}_{\mathsf{Withdraw}})$
15:    return $(\sigma^V_{\mathsf{Assert}}, \sigma^V_{\mathsf{Withdraw}})$
16: end function

17: function VerifySigs$_P$($\mathsf{pk}_V, \mathcal{T}, \mathsf{presigs}_V$)
18:    Parse $\mathsf{presigs}_V = (\sigma^V_{\mathsf{Assert}}, \sigma^V_{\mathsf{Withdraw}})$
19:    Verify $\mathsf{Sig}_{BTC}.\mathtt{Verify}(\mathsf{pk}_V, \overline{\mathsf{tx}}_{\mathsf{Assert}}, \sigma^V_{\mathsf{Assert}})$
20:    Verify $\mathsf{Sig}_{BTC}.\mathtt{Verify}(\mathsf{pk}_V, \overline{\mathsf{tx}}_{\mathsf{Withdraw}}, \sigma^V_{\mathsf{Withdraw}})$
21: end function

22: function VerifySigs$_V$($\mathsf{pk}_P, \mathcal{T}, \mathsf{presigs}_P$)
23:    Parse $\mathsf{presigs}_P = (\sigma^P_{\mathsf{ChallengeAssert}}, \sigma^P_{\mathsf{NoWithdraw}})$
24:    Verify $\mathsf{Sig}_{BTC}.\mathtt{Verify}(\mathsf{pk}_P, \overline{\mathsf{tx}}_{\mathsf{ChallengeAssert}}, \sigma^P_{\mathsf{ChallengeAssert}})$
25:    Verify $\mathsf{Sig}_{BTC}.\mathtt{Verify}(\mathsf{pk}_P, \overline{\mathsf{tx}}_{\mathsf{NoWithdraw}}, \sigma^P_{\mathsf{NoWithdraw}})$
26: end function
```

means that any transaction spending it must be signed by both the Prover and the Verifier. This ensures that neither party can unilaterally withdraw the Bitcoin.

Pre-signed transactions. During the setup phase, the Prover and the Verifier pre-sign the set of allowed transactions that can be posted by the other party during the proving phase. For example, the Verifier pre-signs the Withdraw transaction that uses Deposit's output as an input and gives this signature to the Prover during setup. Since the signatures are on the transaction skeleton, they can be signed during setup without knowing the transaction witnesses. This ensures the following:

- *Unilateral posting*: The Prover can post the Withdraw transaction during the proving phase without having to depend on the Verifier to sign it.
- *Output binding*: The Prover cannot post any other transaction that uses Deposit's output as an input because he does not have the Verifier's signature for such a transaction.
- *Input binding*: The pre-signed transaction Withdraw commits to the hash of a specific Assert transaction skeleton because it uses an output of Assert as an input. This ensures that the Prover cannot post the Withdraw transaction unless he first posts the correct Assert transaction.

In general, these properties ensure that only transactions from the set $\mathcal{T}$ created during the setup phase can be posted during the proving phase.

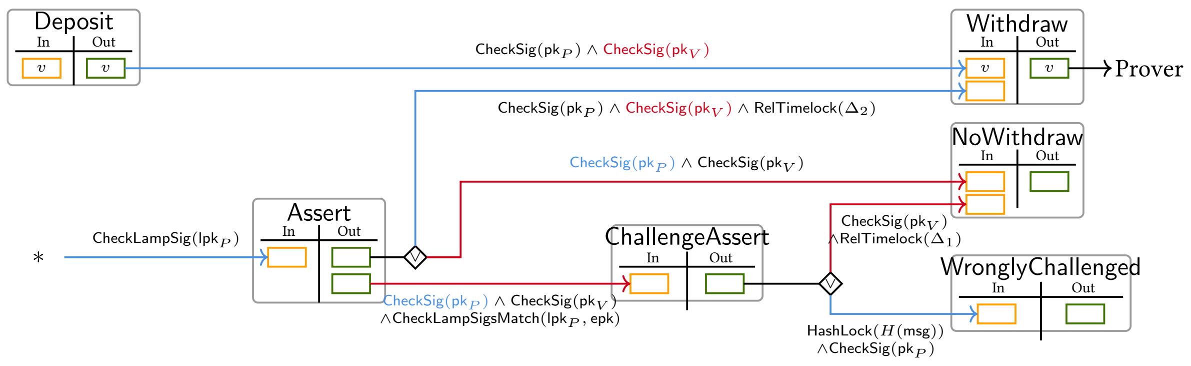


Figure 5: Illustration of the Bitcoin transaction graph. Gray boxes represent transactions. A transaction's inputs and outputs are represented by orange and green boxes, respectively, inside the transaction. Numbers inside the inputs and outputs represent the amount of Bitcoin. Empty boxes indicate the minimum amount required to cover the transaction fees. Arrows connect one transaction's output used as an input by another transaction. The ◇ shape indicates that the transaction output can be used as input by any one out of multiple possible transactions. Locking scripts written on an arrow entering a transaction must be satisfied by the transaction witness. Arrows entering a transaction posted by the Prover are blue and arrows entering a transaction posted by the Verifier are red. A red CheckSig(pk$_V$) indicates that the transaction taking that input is pre-signed by the Verifier during setup. Similarly, a blue CheckSig(pk$_P$) indicates that the transaction is pre-signed by the Prover during setup.

Timelocks. The Withdraw transaction must take as another input the 0-th output of Assert, which carries the condition RelTimelock(Δ_2) in its locking script. This ensures that the Prover cannot post the Withdraw transaction until Assert is posted on chain and the Bitcoin chain has grown by at least Δ_2 blocks, which should give enough time for the Verifier to "stop" a malicious Prover. How does the Verifier "stop" the Prover?

Connector outputs. A connector output is an output of a transaction when there are multiple transactions that could potentially spend it. For example, in Fig. 5, the 0-th output of Assert is a connector output because it can be spent by either the Withdraw or the NoWithdraw transaction. If the Verifier posts a NoWithdraw transaction within Δ_2 blocks of the Assert transaction, he stops the Prover from posting the Withdraw transaction because one of the inputs of the Withdraw transaction is no longer available.

Another connector output, the output of ChallengeAssert, ensures that it can be spent in one of two ways. Either the Prover posts the WronglyChallenged transaction, whose HashLock(H(msg)) script requires the Prover to have decrypted the secret msg. Or, the Verifier posts the NoWithdraw transaction if the Prover doesn't post the WronglyChallenged transaction within Δ_1 blocks of the ChallengeAssert transaction. Posting NoWithdraw stops the Prover from ever posting the Withdraw transaction.

Commitment using Lamport signatures. The Assert transaction is used by the Prover to post π_1 on chain and the ChallengeAssert transaction is used by the Verifier to post the input labels for the proof π_1 on chain. The locking scripts in these transactions' inputs ensure that the Verifier indeed posts input labels for the same proof π_1 that the Prover posts.

The input of the Assert transaction carries a locking scriptCheckLampSig(lpk$_P$) which means the Prover must provide his Lamport signature μ on some message π_1. The script does not care about the message π_1 itself, but only that the Prover has committed to some message.

Conveniently, the input labels of the garbled circuit are exactly the Lamport signatures of the input π_1

under the garbled circuit's encoding key ek. So, the input of the ChallengeAssert transaction requires the Verifier to provide a Lamport signature under the public key epk (corresponding to the encoding key ek). Not only that, the script CheckLampSigsMatch(lpk_P, epk) requires the Verifier to also post a Lamport signature for the same message under the Prover's Lamport key lpk_P. This ensures that the Verifier Lamport signs the same π_1 that the Prover did, since the Verifier cannot forge a Lamport signature under lpk_P for any message other than π_1 without knowing the Prover's Lamport secret key. This approach to matching the labels was first proposed in [Che25].

Putting it all together. Together, the transaction graphs ensure that i) the Prover commits to a proof element π_1, ii) the Verifier posts input labels for the same proof π_1, iii) the Prover can post Withdraw only if he decrypts msg, and iv) the Verifier can post NoWithdraw otherwise. We prove in Sec. 7 that these transactions together with the witness encryption and garbled circuit satisfy the BitVM-core security properties.

6.2 Verifying Setup Correctness

We augment the setup phase of the protocol to allow the Prover and Verifier to verify that the setup was run correctly. This verification is done off-chain either party can abort if the verification fails, without losing any money, because this is done before the Bitcoin is locked.

Verifying setup involves verifying the following:

1. Statement $\boldsymbol{x}$ (deterministic given Prover's public key pk_P)
2. Transactions $\mathcal{T}, \mathcal{S}$ (created deterministically given both parties' Bitcoin public keys, Prover's Lamport public key, hashes of the input labels, and hash of the secret msg)
3. Pre-signatures exchanged between the Prover and the Verifier (verified using the signature verification algorithm)
4. Randomness used for encryption/garbling is independent of the proof element π_1 (required for the condition in Lem. 4). This is satisfied in many use-cases, e.g., when the relation requires a certain transaction to be finalized in Ethereum, the Verifier cannot compute π_1 at setup. Otherwise, this can be achieved by adding some randomness to the Prover's Groth16 witness.
5. Witness encryption ciphertext $\mathsf{ct}_{\mathsf{setup}}$
6. Garbled circuit ciphertext $\mathsf{ct}_{\mathsf{GC}}$ and hashes of the input labels

What remains is to verify that the witness encryption ciphertext $\mathsf{ct}_{\mathsf{setup}}$ and the garbled circuit were computed correctly with respect to the statement $\boldsymbol{x}$ and the secret msg. This verification, carried out off-chain, can be achieved using various techniques from the literature, such as zero-knowledge proofs or cut-and-choose. We adopt cut-and-choose for its relative simplicity and efficiency.

The cut-and-choose setup protocol is shown in Alg. 4 (Prover) and Alg. 5 (Verifier). In this protocol, the Verifier generates N_{CC} instances, each with a secret msg_i, independent randomness r_i, and a garbled circuit for $f_i(\pi) \to r_i\pi$. The Verifier commits to all instances, then theProver randomly selects a subset $\mathcal{I} \subset [N_{\mathsf{CC}}]$ of size M_{CC}. For every instance $i \notin \mathcal{I}$, the Verifier "opens" the instance by revealing the underlying secrets and the randomness used for both encryption and garbling. The Prover then recomputes the corresponding ciphertexts and labels to check that they match the Verifier's commitments. The Prover finalizes the remaining M_{CC} instances by storing their ciphertexts. After verifying correctness for $N_{\mathsf{CC}} - M_{\mathsf{CC}}$ opened instances, the probability that all M_{CC} finalized instances are faulty is at most $p_{\text{err}} = \binom{N_{\mathsf{CC}}}{M_{\mathsf{CC}}}^{-1}$.

In the proving phase (Alg. 6), the Verifier reveals the input labels for all M_{CC} finalized instances and the Prover can withdraw Bitcoin if he successfully decrypts the secret for any one of these M_{CC} instances—an event that succeeds except with probability p_{err}. In Sec. 9, we explore different parameter choices and the trade-offs in setup time, off-chain storage, and on-chain cost. We also discuss possible optimizations to reduce on-chain cost by not posting all input labels on chain.

7 Security Proof

7.1 Security Proof Assuming Honest Setup

To warm up, we prove that the protocol in Algs. 1 and 2 is secure in the proving phase assuming that the setup is run honestly by both Prover and Verifier. First, we define this honest-setup security.

Definition 16 (Honest-Setup u-Robustness). *For all NP relations $\mathcal{R}$, all* PPT *adversarial Verifiers V^*, all rounds $r \in \mathbb{N}$, the following holds:*

$$\Pr\left[\begin{array}{l|l} (b=1) & \mathsf{crs} \leftarrow \mathtt{Gen}(\mathcal{R}) \\ \wedge\,(\exists\, \mathsf{tx} \in \mathcal{S} \text{ s.t. } : & (\boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P, \mathsf{st}_V) \leftarrow \mathsf{out}\langle P_{\mathtt{Setup}}(\mathsf{crs}), V_{\mathtt{Setup}}(\mathsf{crs})\rangle \\ \mathsf{tx} \in \mathcal{L}_P^{r+u}) & b \leftarrow \mathsf{out}_P(\langle P_{\mathtt{Prove}}(\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P, \boldsymbol{w}), V^*(\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_V)\rangle_r) \\ & (\boldsymbol{x}, \boldsymbol{w}) \in \mathcal{R} \end{array}\right] \geqslant \begin{array}{l} 1-2^{-\kappa} \\ -\,\mathsf{negl}(\lambda) \end{array} \quad (30)$$

Definition 17 (Honest-Setup Knowledge Soundness). *For all NP relations $\mathcal{R}$, all* PPT *adversarial Provers P^*, there exists a* PPT *extractor $\mathcal{E}$ such that for every benign auxiliary input* $\mathsf{aux} \in \{0,1\}^{\mathsf{poly}(\lambda)}$*:*

$$\Pr\left[\begin{array}{l|l} & \mathsf{crs} \leftarrow \mathtt{Gen}(\mathcal{R}) \\ (\boldsymbol{x}, \mathcal{E}(\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, & (\boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P, \mathsf{st}_V) \leftarrow \mathsf{out}\langle P_{\mathtt{Setup}}(\mathsf{crs}), V_{\mathtt{Setup}}(\mathsf{crs})\rangle \\ \mathsf{st}_P, T, \mathsf{aux})) \in \mathcal{R} \quad : & T \leftarrow \langle P^*(\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P, \mathsf{aux}), V_{\mathtt{Prove}}(\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_V)\rangle \\ & \exists\, \tau \in \mathbb{N},\ \exists\, \mathsf{tx} \in \mathcal{S},\ \exists \text{ honest } H : \mathsf{tx} \in \mathcal{L}_H^{\tau} \end{array}\right] \geqslant \begin{array}{l} 1-2^{-\kappa} \\ -\,\mathsf{negl}(\lambda) \end{array} \quad (31)$$

Note that the difference with respect to the BitVM-core security definitions (Sec. 3) is that setup is run by $P_{\mathtt{Setup}}$ and $V_{\mathtt{Setup}}$ instead of an adversarial Prover P^* or Verifier V^*. Other than that, we provide the outputs of the setup phase to the adversarial Prover P^* or Verifier V^* in the proving phase since they were not participating in the setup phase. In the case of honest setup, the setup doesn't abort by definition.

7.1.1 Proof of Honest-Setup u-Robustness

Theorem 6. *Assuming $\mathcal{F}_{\mathsf{BTC}}$ satisfies safety, u_{BTC}-liveness, and (τ, s)-chain growth for $s > u_{BTC}$, the protocol in Algs. 1 and 2 with $\Delta_1 > 2u_{BTC}$ and $\Delta_2 > u_{BTC}$ satisfies honest-setup u-robustness where $u = \tau^{-1}(\Delta_2 + u_{BTC})$.*

Proof. For a given NP relation $\mathcal{R}$, let $\mathsf{crs} \leftarrow \mathtt{Groth16.Gen}(\mathcal{R})$ and let $\boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P, \mathsf{st}_V$ be generated as per Alg. 1, where $\mathcal{S} = \{\mathsf{tx}_{\mathsf{Withdraw}}\}$ (see Alg. 3 line 4). Suppose the Prover runs $P_{\mathtt{Prove}}$ as per Alg. 2 starting at round r.

We go through the steps of the $P_{\mathtt{Prove}}$ algorithm and show that with overwhelming probability, $\mathsf{tx}_{\mathsf{Withdraw}} \in \mathcal{L}_P^{r+u}$, and we calculate the value u.

Alg. 2 line 3: Prover generates (π_1, π_2, π_3). By perfect correctness of Groth16 (Thm. 2, Def. 1), this is a valid proof, i.e., $\mathtt{Verify}(\mathsf{crs}, \boldsymbol{x}, (\pi_1, \pi_2, \pi_3)) = 1$.

Alg. 2 line 5: Prover calls $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{Assert}})$ at round r when $\mathcal{L}_P^r$ has height h_0. $\mathsf{tx}_{\mathsf{Assert}}$ is valid because it contains a valid Lamport signature for the key lpk_P (Alg. 2 line 4). It is also unstoppable with respect to the state st_V because st_V does not contain the Lamport signing key lsk_P. Therefore, due to liveness, $\mathsf{tx}_{\mathsf{Assert}} \in \mathcal{L}_P[: h_1]$ where $h_1 \leqslant h_0 + u_{BTC}$.

Case 1: $\mathsf{tx}_{\mathsf{ChallengeAssert}} \notin \mathcal{L}_P[: h_1 + \Delta_2 - u_{BTC}]$ for all rounds. When $h(\mathcal{L}_P) = h_1 + \Delta_2$, Prover calls $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{Withdraw}})$ (Alg. 2 line 7). $\mathsf{tx}_{\mathsf{Withdraw}}$ is valid because i) its parents[17] $\mathsf{tx}_{\mathsf{Deposit}}$ and $\mathsf{tx}_{\mathsf{Assert}}$ are in the ledger, ii) its transaction witness contains the Verifier's pre-signature obtained during the setup and the Prover's signature (Alg. 2 line 6), and iii) the timelock on its input (1) has expired. Moreover, $\mathsf{tx}_{\mathsf{Withdraw}}$ is u_{BTC}-unstoppable when $\Delta_1 > 2u_{BTC}$ because i) $\mathsf{tx}_{NoPayout}$ is not valid because if $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ was included by the adversary at height $> h_1 + \Delta_2 - u_{BTC}$, the timelock on input (1) of $\mathsf{tx}_{\mathsf{NoWithdraw}}$ would not expire by height $h_1 + \Delta_2 + u_{BTC}$, and ii) the adversary (not knowing sk_P) cannot produce a transaction witness for any other transaction spending either input of $\mathsf{tx}_{\mathsf{Withdraw}}$. Therefore, due to liveness, $\mathsf{tx}_{\mathsf{Withdraw}} \in \mathcal{L}_P[: h_2]$ where $h_2 \leqslant h_1 + \Delta_2 + u_{BTC}$.

Case 2: $\mathsf{tx}_{\mathsf{ChallengeAssert}} \in \mathcal{L}_P[h_3]$ for some $h_3 < h_1 + \Delta_2 - u_{BTC}$ at some round.

1. Upon seeing $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ in $\mathcal{L}_P$, Prover extracts the input labels posted by the Verifier in $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ (Alg. 2 line 9). For $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ to be valid, its transaction witness must satisfy the script $\mathsf{CheckLampSigsMatch}(\mathsf{lpk}_P, \mathsf{ek})$ which requires a valid Lamport signature under the Prover's key lpk_P and a valid Lamport signature under the Verifier's key ek for the same message. Since the adversary does not know lsk_P, he cannot produce a valid Lamport signature under lpk_P for any message other than π_1. Therefore, the extracted labels satisfy $L \leftarrow \mathsf{Encode}(\mathsf{ek}, \pi_1)$.

2. Prover evaluates the garbled circuit (Alg. 2 line 10) and due to correctness of the garbled circuit (Def. 3), $\mathsf{Eval}(\mathsf{ct}_{\mathsf{GC}}, L) = \mathsf{Enc}_{\mathsf{prove}}(\mathsf{crs}, \pi_1; r)$ (Constr. 1).

3. Prover decrypts the message (Alg. 2 line 11) and due to correctness of the witness encryption scheme, the Prover learns msg.

4. Prover constructs the transaction witness for the $\mathsf{WronglyChallenged}$ transaction (Alg. 2 line 12). This transaction is valid because i) its parent $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ is in the ledger, and ii) its transaction witness contains the message msg. This transaction is u_{BTC}-unstoppable when $\Delta_1 > u_{BTC}$ because the adversary does not know sk_P and the timelock Δ_1 on its input (0) does not expire in u blocks. Therefore, due to liveness, $\mathsf{tx}_{\mathsf{WronglyChallenged}} \in \mathcal{L}_P[: h_4]$ where $h_4 \leqslant h_3 + u_{BTC}$.

5. This guarantees that for all $h > h_4$, $\mathsf{tx}_{\mathsf{NoWithdraw}} \notin \mathcal{L}_P[: h]$ because input (1) of $\mathsf{tx}_{\mathsf{NoWithdraw}}$ is no longer available. This holds in particular for $h = h_1 + \Delta_2$.

6. The Prover calls $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{Withdraw}})$ when $h(\mathcal{L}_P) = h_1 + \Delta_2$. Following the same arguments as in Case 1, this transaction is valid and unstoppable, therefore by liveness, $\mathsf{tx}_{\mathsf{Withdraw}} \in \mathcal{L}_P^{[} : h_2]$ where $h_2 \leqslant h_1 + \Delta_2 + u_{BTC}$.

Due to the chain growth property (Thm. 3), $\mathsf{tx}_{\mathsf{Withdraw}} \in \mathcal{L}_P^{r+u}$ where $u = \tau^{-1}(\Delta_2 + u_{BTC})$.

□

7.1.2 Proof of Honest-Setup Knowledge Soundness

To prove knowledge soundness, we first combine the witness encryption scheme Constr. 1 and the garbled circuit Constr. 2 and prove that no adversary given the ciphertexts of both and the input labels of the garbled circuit can decrypt the message without knowing a valid witness for the relation $\mathcal{R}$.

[17]A transaction tx is a parent of another transaction tx′ one of the inputs of tx′ is an output of tx.

Lemma 10. *Let* $\mathtt{Gen}, \mathtt{Enc}_{\mathsf{setup}}, \mathtt{Enc}_{\mathsf{prove}}, \mathsf{Dec}$ *(Constr. 1) be an adaptively secure witness encryption scheme (Def. 12) and let* $\mathsf{Garble}, \mathsf{Encode}, \mathsf{Eval}$ *(Constr. 2) be adaptively private (Def. 3). Then in the random oracle model, for all NP relations* $\mathcal{R}$*, for all* PPT *adversaries* $\mathcal{A}$*, there exists a* PPT *extractor* $\mathcal{E}$ *such that if*

$$\Pr\left[\mathsf{Hash}_{BTC}(\mathsf{msg}') = h_{\mathsf{msg}} : \begin{array}{l} \mathsf{crs} \leftarrow \mathtt{Gen}(\mathcal{R}) \\ \boldsymbol{x} \leftarrow \mathcal{A}(\mathsf{crs}) \\ \mathsf{msg} \leftarrow\$ \mathcal{M} \\ r \leftarrow\$ \mathbb{F}_q^* \\ \mathsf{ct}_{\mathsf{setup}} \leftarrow \mathtt{Enc}_{\mathsf{setup}}(\mathsf{crs}, \boldsymbol{x}, \mathsf{msg}; r) \\ \mathsf{ct}_{\mathsf{GC}}, \mathsf{ek} \leftarrow \mathsf{Garble}(r) \\ h_{\mathsf{msg}} \leftarrow \mathsf{Hash}_{BTC}(\mathsf{msg}) \\ \mathsf{epk}_j^b \leftarrow \mathsf{Hash}_{BTC}(\mathsf{ek}_j^b) \quad \forall j \in \{1, \ldots, 2n\}, b \in \{0,1\} \\ \pi_1 \leftarrow \mathcal{A}(\mathsf{ct}_{\mathsf{setup}}, \mathsf{ct}_{\mathsf{GC}}, h_{\mathsf{msg}}, \mathsf{epk}) \\ L \leftarrow \mathsf{Encode}(\mathsf{ek}, \pi_1) \\ \mathsf{msg}' \leftarrow \mathcal{A}(L) \end{array}\right] = \epsilon$$

then

$$\Pr\left[(\boldsymbol{x}, \boldsymbol{w}) \in \mathcal{R} : \boldsymbol{w} \leftarrow \mathcal{E}(\mathsf{crs}, \boldsymbol{x}, \mathsf{ct}_{\mathsf{setup}}, \mathsf{ct}_{\mathsf{GC}}, h_{\mathsf{msg}}, \mathsf{epk}, L)\right] \geq \epsilon - \mathsf{negl}(\lambda)$$

Proof. We reduce the lemma to the adaptive privacy of the garbling scheme (Def. 3) and to the adaptive security of witness encryption (Def. 12) via a hybrid argument.

Games. Call the game in the lemma statement G_0. The adversary wins G_0 when $\mathsf{Hash}_{BTC}(\mathsf{msg}') = h_{\mathsf{msg}}$, where msg' is the adversary's output and $h_{\mathsf{msg}} = \mathsf{Hash}_{BTC}(\mathsf{msg})$. Let the success probability of the adversary in G_0 be $p_0 = \epsilon$.

Game G_1 (garbled circuit replaced by simulator): same as G_0 except we replace the real garbling by the simulator. Let C be the circuit from the construction (on input π_1 it outputs $\mathsf{ct}_{\mathsf{prove}} = r\pi_1$). Run $\mathsf{ct}_{\mathsf{GC}} \leftarrow \mathsf{Sim}_1(\mathsf{topo}(C))$, sample $\mathsf{epk} \leftarrow\$ \left(\{0,1\}^\lambda\right)^{2n}$, and set $L \leftarrow \mathsf{Sim}_2(r\pi_1, \mathsf{msg})$. The simulator Sim_1 is assumed to pass state to Sim_2. The win condition remains $\mathsf{Hash}_{BTC}(\mathsf{msg}') = h_{\mathsf{msg}}$. The simulator also programs the random oracle Hash_{BTC} such that if π is represented as $(x \in \{0,1\}^n, y \in \{0,1\}^n)$, then for all $j = 1, \ldots, n$, $\mathsf{epk}_j^{x_j} = \mathsf{Hash}_{BTC}(L_j)$ and $\mathsf{epk}_{n+j}^{y_j} = \mathsf{Hash}_{BTC}(L_{n+j})$.

$$p_1 = \Pr\left[\mathsf{Hash}_{BTC}(\mathsf{msg}') = h_{\mathsf{msg}} : \begin{array}{l} \mathsf{crs} \leftarrow \mathtt{Groth16.Gen}(\mathcal{R}) \\ \boldsymbol{x} \leftarrow \mathcal{A}(\mathsf{crs}) \\ \mathsf{msg} \leftarrow\$ \mathcal{M} \\ r \leftarrow\$ \mathbb{F}_q^* \\ \mathsf{ct}_{\mathsf{setup}} \leftarrow \mathtt{Enc}_{\mathsf{setup}}(\mathsf{crs}, \boldsymbol{x}, \mathsf{msg}; r) \\ h_{\mathsf{msg}} \leftarrow \mathsf{Hash}_{BTC}(\mathsf{msg}) \\ \mathsf{ct}_{\mathsf{GC}} \leftarrow \mathsf{Sim}_1(\mathsf{topo}(C)) \\ \mathsf{epk} \leftarrow\$ \left(\{0,1\}^\lambda\right)^{2n} \\ \pi_1 \leftarrow \mathcal{A}(\mathsf{ct}_{\mathsf{setup}}, \mathsf{ct}_{\mathsf{GC}}, h_{\mathsf{msg}}, \mathsf{epk}) \\ L \leftarrow \mathsf{Sim}_2(\pi_1, r\pi_1) \\ \mathsf{msg}' \leftarrow \mathcal{A}(L) \end{array}\right]$$

$G_0 \approx G_1$ **(indistinguishability by garbling).** By the adaptive privacy of the garbling scheme (Def. 3) and the programmability of the random oracle, for every PPT adversary the difference in the probability that the adversary's output satisfies the win condition is at most $\mathsf{negl}(\lambda)$. Thus $\left|p_0 - p_1\right| \leq \mathsf{negl}(\lambda)$, so $p_1 \geq \epsilon - \mathsf{negl}(\lambda)$.

Game G_2 (garbling replaced by actual circuit evaluation). Instead of giving the adversary the garbled circuit $\mathsf{ct}_{\mathsf{GC}}$, encoding public key epk, and the input labels L, the adversary chooses the proof element π_1 and is directly given the evaluation of the garbled circuit, i.e., $\mathsf{ct}_{\mathsf{prove}} = r\pi$.

$$p_2 = \Pr\left[\mathsf{Hash}_{BTC}(\mathsf{msg}') = h_{\mathsf{msg}} : \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{Groth16.Gen}(\mathcal{R}) \\ \boldsymbol{x} \leftarrow \mathcal{A}_2(\mathsf{crs}) \\ \mathsf{msg} \leftarrow\$ \mathcal{M} \\ r \leftarrow\$ \mathbb{F}_q^* \\ \mathsf{ct}_{\mathsf{setup}} \leftarrow \mathsf{Enc}_{\mathsf{setup}}(\mathsf{crs}, \boldsymbol{x}, \mathsf{msg}; r) \\ h_{\mathsf{msg}} \leftarrow \mathsf{Hash}_{BTC}(\mathsf{msg}) \\ \pi_1 \leftarrow \mathcal{A}_2(\mathsf{ct}_{\mathsf{setup}}, h_{\mathsf{msg}}) \\ \mathsf{ct}_{\mathsf{prove}} \leftarrow \mathsf{Enc}_{\mathsf{prove}}(\mathsf{crs}, \pi_1; r) \\ \mathsf{msg}' \leftarrow \mathcal{A}_2(\mathsf{ct}_{\mathsf{prove}}) \end{array}\right]$$

$G_1 \approx G_2$. Since the adversary knows the circuit topology $\mathsf{topo}(C)$, it can run Sim_1 to simulate $\mathsf{ct}_{\mathsf{GC}}$ and it can also sample epk randomly as was done in game G_1. Given $r\pi_1$, the adversary can also run Sim_2 to simulate the labels L. Therefore, there exists a PPT $\mathcal{A}_2$ such that $p_2 = p_1$.

Game G_3 is the adaptive witness encryption security game from Def. 12.

$$p_3 = \Pr\left[b = b' : \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{Groth16.Gen}(\mathcal{R}) \\ (\boldsymbol{x}, \mathsf{msg}_0, \mathsf{msg}_1) \leftarrow \mathcal{A}_3(\mathsf{crs}) \\ b \leftarrow\$ \{0,1\} \\ r \leftarrow\$ \mathbb{F}_q^* \\ \mathsf{ct}_{\mathsf{setup}} \leftarrow \mathsf{Enc}_{\mathsf{setup}}(\mathsf{crs}, \boldsymbol{x}, \mathsf{msg}_b; r) \\ \pi_1 \leftarrow \mathcal{A}_3(\mathsf{ct}_{\mathsf{setup}}) \\ \mathsf{ct}_{\mathsf{prove}} \leftarrow \mathsf{Enc}_{\mathsf{prove}}(\mathsf{crs}, \pi_1; r) \\ b' \leftarrow \mathcal{A}_3(\mathsf{ct}_{\mathsf{prove}}) \end{array}\right]$$

Step 3: Reduction from G_2 to G_3 (WE game). Given a PPT adversary $\mathcal{A}_2$ that wins Game G_2 with probability $p_2 \geq \epsilon - \mathsf{negl}(\lambda)$, we construct a PPT adversary $\mathcal{A}_3$ for Game G_3. In the random oracle model, $\mathcal{A}_3$ and the WE challenger share the same random oracle Hash_{BTC}.

Construction of $\mathcal{A}_3$ from $\mathcal{A}_2$.

(i) **On input** crs: Run $\boldsymbol{x} \leftarrow \mathcal{A}_2(\mathsf{crs})$. Sample $\mathsf{msg}_0, \mathsf{msg}_1 \leftarrow\$ \mathcal{M}$ uniformly. Output $(\boldsymbol{x}, \mathsf{msg}_0, \mathsf{msg}_1)$ to the WE challenger.

(ii) **On input** $\mathsf{ct}_{\mathsf{setup}}$**:** The challenger has computed $\mathsf{ct}_{\mathsf{setup}} \leftarrow \mathsf{Enc}_{\mathsf{setup}}(\mathsf{crs}, \boldsymbol{x}, \mathsf{msg}_b; r)$ for unknown $b \in \{0,1\}$. In G_2, $\mathcal{A}_2$ receives $(\mathsf{ct}_{\mathsf{setup}}, h_{\mathsf{msg}})$ with $h_{\mathsf{msg}} = \mathsf{Hash}_{BTC}(\mathsf{msg})$; here the message is msg_b, so the correct hash is $\mathsf{Hash}_{BTC}(\mathsf{msg}_b)$, but we do not know b. Guess the challenge bit: set $h := \mathsf{Hash}_{BTC}(\mathsf{msg}_0)$. Run $\pi_1 \leftarrow \mathcal{A}_2(\mathsf{ct}_{\mathsf{setup}}, h)$ and output π_1 to the challenger.

(iii) **On input** $\mathsf{ct}_{\mathsf{prove}}$**:** Run $\mathsf{msg}' \leftarrow \mathcal{A}_2(\mathsf{ct}_{\mathsf{prove}})$. In G_2, when $\mathcal{A}_2$ wins it outputs msg' with $\mathsf{Hash}_{BTC}(\mathsf{msg}') = h = \mathsf{Hash}_{BTC}(\mathsf{msg}_0)$. Set $b' := 0$ if $\mathsf{Hash}_{BTC}(\mathsf{msg}') = \mathsf{Hash}_{BTC}(\mathsf{msg}_0)$, $b' \leftarrow\$ \{0,1\}$ otherwise. Output b' to the challenger.

Analysis.

1. Case $b = 0$: When $b = 0$, we set $h = \mathsf{Hash}_{BTC}(\mathsf{msg}_0)$, so the view of $\mathcal{A}_2$ matches G_2. With probability p_2, $\mathcal{A}_2$ outputs msg' with $\mathsf{Hash}_{BTC}(\mathsf{msg}') = h$, in which case $\mathcal{A}_3$ outputs $b' = 0 = b$. With probability $1 - p_2$, $\mathcal{A}_2$ outputs msg' with $\mathsf{Hash}_{BTC}(\mathsf{msg}') \neq h$, in which case $\Pr[b' = b] = \frac{1}{2}$. Thus, $\Pr[b' = b | b = 0] = p_2 + \frac{1}{2}(1 - p_2) = \frac{1}{2} + \frac{p_2}{2}$.

2. Case $b = 1$: When $b = 1$, we set $h = \mathsf{Hash}_{BTC}(\mathsf{msg}_0)$, but the encrypted message is msg_1. Since Hash_{BTC} is a random oracle and msg_0 and msg_1 are chosen independently, $\Pr[\mathsf{Hash}_{BTC}(\mathsf{msg}') = h] = \mathsf{negl}(\lambda)$. When $\mathsf{Hash}_{BTC}(\mathsf{msg}') \neq h$, $\Pr[b' = b] = \frac{1}{2}$. Thus, $\Pr[b' = b | b = 1] = (1 - \mathsf{negl}(\lambda))\frac{1}{2}$.

Putting the cases together, we get $\Pr[b' = b] = \frac{1}{2}\left(\frac{1}{2} + \frac{p_2}{2}\right) + \frac{1}{2}(1 - \mathsf{negl}(\lambda)) = \frac{1}{2} + \frac{p_2}{4} - \mathsf{negl}(\lambda)$.

Step 4: Extraction. By the adaptive security of witness encryption, there exists a PPT extractor $\mathcal{E}'$ such that

$$\Pr\left[\left((\mathsf{crs}, \boldsymbol{x}, \pi_1), \boldsymbol{w}\right) \in \mathcal{R}' : \boldsymbol{w} \leftarrow \mathcal{E}'(\mathsf{crs}, \boldsymbol{x}, \pi_1)\right] \geq \epsilon' - \mathsf{negl}(\lambda) \geq \frac{p_2}{4} - \mathsf{negl}(\lambda) \leqslant \epsilon - \mathsf{negl}(\lambda),$$

A witness for $\mathcal{R}'$ is also a witness for $\mathcal{R}$ (see Eq. (24)). The lemma's extractor $\mathcal{E}$ has the view of the lemma's adversary $\mathcal{A}$, which includes $\mathsf{crs}, \boldsymbol{x}, \pi_1$, and thus can run $\mathcal{E}'$ and succeed with probability $\epsilon' - \mathsf{negl}(\lambda)$. □

Theorem 7. *Assuming $\mathcal{F}_{\mathsf{BTC}}$ satisfies safety, u_{BTC}-liveness, and (τ, s)-chain growth for $s > u_{BTC}$, the protocol in Algs. 1 and 2 with $\Delta_2 > \Delta_1 + 2u_{BTC}$ satisfies honest-setup knowledge soundness.*

Proof. For a given NP relation $\mathcal{R}$, let $\mathsf{crs} \leftarrow \mathsf{Groth16.Gen}(\mathcal{R})$ and let $\boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P, \mathsf{st}_V$ be generated as per Alg. 1, where $\mathcal{S} = \{\mathsf{tx}_{\mathsf{Withdraw}}\}$ (see Alg. 3 line 4). Suppose the Verifier runs $V_{\mathtt{Prove}}$ as per Alg. 2. Suppose that at some round r and some honest party H, $\mathsf{tx}_{\mathsf{Withdraw}} = \mathcal{L}_H^r[h]$.

For $\mathsf{tx}_{\mathsf{Withdraw}}$ to be valid, its ancestors $\mathsf{tx}_{\mathsf{Assert}}$ and $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ must be in the ledger. Suppose $\mathsf{tx}_{\mathsf{Assert}} = \mathcal{L}_H^r[h_0]$. Due to the timelock on input (1) of $\mathsf{tx}_{\mathsf{Withdraw}}$, $\mathsf{tx}_{\mathsf{Assert}}$ must have been included at least Δ_2 blocks earlier, i.e., $h_0 \leqslant h - \Delta_2$.

The Verifier, upon seeing $\mathsf{tx}_{\mathsf{Assert}} \in \mathcal{L}_V[h_0]$ at some round, called $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{ChallengeAssert}})$ (Alg. 2 line 20). This transaction is valid because its transaction witness contains both parties' signatures, the Prover's Lamport signature and the Verifier's Lamport signature for the same message π_1 (Alg. 2 line 19). This transaction is unstoppable because the adversary does not know ek or sk_V. Due to liveness, $\mathsf{tx}_{\mathsf{ChallengeAssert}} \in \mathcal{L}_V[: h_1]$ where $h_1 \leqslant h_0 + u_{BTC}$.

When $h(\mathcal{L}_V) = h_1 + \Delta_1$, the Verifier calls $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{NoWithdraw}})$ (Alg. 2 line 22). This transaction is valid because its transaction witness contains the Prover's pre-signature obtained during the setup and the Verifier's signature (Alg. 2 line 21), and the timelock Δ_1 on its input (1) has expired.

Case 1: $\mathsf{tx}_{\mathsf{NoWithdraw}} \in \mathcal{L}_H[: h_2]$ for $h_2 = h_1 + \Delta_1 + u_{BTC} \leqslant h + \Delta_1 + 2u_{BTC} - \Delta_2$. Since $\Delta_2 > \Delta_1 + 2u_{BTC}$, $h_2 < h$. This is a contradiction because if $\mathsf{tx}_{\mathsf{NoWithdraw}} \in \mathcal{L}_H[: h_2]$, and $\mathsf{tx}_{\mathsf{NoWithdraw}} \in \mathcal{L}_H[: h]$, then the ledger $\mathcal{L}_H$ is invalid because both transactions contain a common input.

Case 2: $\mathsf{tx}_{\mathsf{NoWithdraw}} \notin \mathcal{L}_H[: h_2]$ for $h_2 = h_1 + \Delta_1 + u_{BTC}$. Due to liveness, this must mean that $\mathsf{tx}_{\mathsf{NoWithdraw}}$ is not u_{BTC}-unstoppable with respect to the ledger $\mathcal{L}_V[h_1 + \Delta_1]$ and the adversary's state $\mathsf{st} = (\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P, \mathsf{aux})$ for some auxiliary input aux. In particular, the adversary must have created a sequence of blocks such that $\mathsf{tx}_{\mathsf{NoWithdraw}}$ is invalid when placed in one of the blocks. Since the timelock Δ_1 has already expired, the adversary must have created a sequence of blocks containing a valid transaction tx' which shares an input with $\mathsf{tx}_{\mathsf{NoWithdraw}}$. The adversary cannot create any other transaction spending output (0) of $\mathsf{tx}_{\mathsf{Assert}}$ (which is input (0) of $\mathsf{tx}_{\mathsf{NoWithdraw}}$) because i) given $\Delta_2 > \Delta_1 + u_{BTC}$, the timelock Δ_2 will not expire by height $h_1 + \Delta_1 + u_{BTC}$ and ii) the adversary does not know sk_V and so cannot produce a valid signature for any transaction other than $\mathsf{tx}_{\mathsf{Withdraw}}$. The adversary cannot create any other transaction spending the output of $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ (which is input (1) of $\mathsf{tx}_{\mathsf{NoWithdraw}}$) through the leaf $\mathsf{CheckSig}(\mathsf{pk}_V) \wedge \mathsf{RelTimelock}(\Delta_1)$ because the adversary does not know sk_V. Therefore, the adversary can stop $\mathsf{tx}_{\mathsf{NoWithdraw}}$ only by spending the output of $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ through the leaf $\mathsf{HashLock}(\mathsf{Hash}_{BTC}(\mathsf{msg})) \wedge \mathsf{CheckSig}(\mathsf{pk}_P)$. The transaction witness of such a transaction must contain

msg' such that $\mathsf{Hash}_{BTC}(\mathsf{msg}') = \mathsf{Hash}_{BTC}(\mathsf{msg})$. Subsequently, from Lem. 10, there exists an extractor $\mathcal{E}$ who has the same view as P^* and can extract $\boldsymbol{w}$ such that $(\boldsymbol{x}, \boldsymbol{w}) \in \mathcal{R}$ with probability close to 1. □

Theorem 8. *Assuming $\mathcal{F}_{\mathsf{BTC}}$ satisfies safety, u_{BTC}-liveness, and (τ, s)-chain growth for $s > u_{BTC}$, the protocol in Algs. 1 and 2 with $\Delta_1 > 2u_{BTC}$ and $\Delta_2 > \Delta_1 + 2u_{BTC}$ satisfies honest-setup knowledge soundness and honest-setup u-robustness where $u = \tau^{-1}(\Delta_2 + u_{BTC})$.*

Proof. From Thm. 7 and Thm. 6. □

8 Extensions and Optimizations

8.1 Multiple Verifiers and Provers

In this paper, we defined BitVM-core as a two-party protocol. The BitVM-core definition and the BABE protocol can be easily extended to capture multiple Verifiers (as in BitVM2-core [LAA+25]) so that soundness holds as long as at least one Verifier is honest. Similarly, BABE can support multiple Provers so that the first Prover to generate a valid proof for his statement can withdraw the Bitcoin.

8.2 Optimistic Path

Leveraging techniques from BitVM2 [LAA+25], BABE can incorporate an optimistic path that bypasses the full proving phase when the Verifier can independently confirm the statement. For instance, if the statement is "Bob repaid his loan on Ethereum", the Verifier can directly check the finalized repayment transaction on the Ethereum blockchain. In such situations, neither a proof from the Prover nor the publication of input labels by the Verifier is necessary on Bitcoin, thus significantly reducing on-chain costs.

9 Evaluation

9.1 Honest Setup

We begin with the metrics in Fig. 2 for the honest-setup protocol. We also justify the values used for BitVM3 in Fig. 2.

On-chain cost. The BABE on-chain cost in Fig. 2 is based on the dispute-path transactions Assert, ChallengeAssert, and WronglyChallenged that are directly responsible for the on-chain proof verification, using the vSizes reported in Tab. 1. This is the basis for the \$56.90 on-chain cost entry reported for BABE in Fig. 2. We estimate USD cost using 1 sat ≈ 0.000955 USD (i.e., 1 BTC ≈ 95,500 USD).

Table 1: On-chain cost for the honest-setup experiment in Fig. 2

Transaction	vSize (vB)	Fee (sat)	Feerate (sat/vB)	USD
Assert	9,240	20,611	2.23	19.68
ChallengeAssert	17,400	38,673	2.22	36.93
WronglyChallenged	149	302	2.02	0.29

The BitVM3 on-chain cost entry in Fig. 2 is based on the corresponding dispute-path transactions from a previously published BitVM3 on-chain experiment, normalized to the same fee rate as the BABE experiment.

Off-chain runtime breakdown. The setup time ($174.90\,\mathrm{ms}$) and decryption time ($126.53\,\mathrm{ms}$) reported for BABE in Fig. 2 are single-instance point estimates, according to our measurements. The detailed breakdown numbers below come from a representative run whose totals differ from these point estimates by at most $\approx 1.5\%$. To justify where this time goes, we decompose the dominant off-chain paths into protocol subroutines that correspond to steps in Algs. 1 and 2 and the constructions in Constr. 1 and Sec. 5.6.

On the setup path, the benchmark totals $177.398\,\mathrm{ms}$ and is dominated by generation of the DRE-selection tables used for the DRE encoding step in Sec. 5.6. ($136.896\,\mathrm{ms}$, 77.17%).

The remaining setup components are the privacy-free Boolean gadget at $16.808\,\mathrm{ms}$ (9.47%), that validates the curve point and derives the feature bits (Sec. 5.6). plus $18.029\,\mathrm{ms}$ (10.16%) of garbling overhead and $5.665\,\mathrm{ms}$ (3.19%) of non-garbling work (WE setup, Lamport-key hashes, and message commitments).

On the decryption path, the benchmark totals $125.834\,\mathrm{ms}$ and is dominated by evaluation of the garbled circuit (Alg. 2 and Sec. 5.6). ($111.538\,\mathrm{ms}$, 88.64%), plus $8.888\,\mathrm{ms}$ (7.06%) of overhead from label conversions/packing.

Off-chain storage (ciphertext size). The BABE storage entry in Fig. 2 is the serialized size of the per-instance off-chain setup artifact that the Prover retains through the protocol (cf. Sec. 6): the witness-encryption setup ciphertext $\mathsf{ct}_{\mathsf{setup}}$ from Constr. 1, the garbled-circuit ciphertext $\mathsf{ct}_{\mathsf{GC}}$ from Constr. 2 (used to derive $\mathsf{ct}_{\mathsf{prove}}$), and the associated hashes/commitments referenced by the Bitcoin scripts. Using compressed canonical serialization of the implementation artifact, this totals $22.16\,\mathrm{MiB}$, according to our calculations.

The witness-encryption ciphertexts $(\mathsf{ct}_{\mathsf{setup}}, \mathsf{ct}_{\mathsf{prove}})$ contribute $480\,\mathrm{B}$. The garbled-circuit ciphertext $\mathsf{ct}_{\mathsf{GC}}$ dominates and splits into $6.35\,\mathrm{MiB}$ for the Boolean-Yao gadgets that validate π_1 and derive the feature vector $\overline{\boldsymbol{u}}(\pi_1)$ (Sec. 5) and $15.77\,\mathrm{MiB}$ for the DRE-selection tables implementing the DRE encoding (Def. 14 and Thm. 4); the remaining $\approx 38\,\mathrm{KB}$ are Lamport-key hashes and message commitments plus serialization overhead.

The $15.77\,\mathrm{MiB}$ DRE term matches the sparsity-aware estimate: for $n = 254$ (Sec. 5), the DRE encoding step stores

$$n\big((3n+1) + (4n+1) + (n+1)\big) \;=\; n(8n+3) \;=\; 516{,}890$$

serialized field elements (Jacobian X, Y, Z with $(3n+1,\ 4n+1,\ n+1)$ field elements per bit). With 32-byte canonical serialization per field element, this is

$$516{,}890 \times 32 \;=\; 16{,}540{,}480\ \mathrm{B} \;\approx\; 15.77\,\mathrm{MiB}.$$

BitVM3 reference point (ciphertext size). The BitVM3 storage entry in Fig. 2 follows from the reported 2.7 billion non-free gates [Bit25b]; using half-gates garbling and interpreting this count as half-gates (one 16-byte ciphertext per non-free gate), this yields $2.7 \times 10^9 \times 16\,\mathrm{B} \approx 41{,}200\,\mathrm{MiB}$.

9.2 Cut-and-Choose Setup Verification

We use cut-and-choose to verify setup correctness against a malicious setup generator. We refer to Algs. 4 and 5 and App. B for the protocol definition.

Off-chain setup cost. Tab. 2 reports setup cost between two protocol roles (Prover and Verifier) executed on a single machine for different $(N_{\mathsf{CC}}, M_{\mathsf{CC}})$ choices; timings exclude network latency. We additionally report peak RAM usage and a breakdown of garbling and evaluation components.[18]

[18]These components correspond to the generation and use of the full protocol ciphertexts: in each cut-and-choose instance the Verifier produces the witness-encryption setup ciphertext $\mathsf{ct}_{\mathsf{setup}}$ (Constr. 1) and the garbled-circuit ciphertext $\mathsf{ct}_{\mathsf{GC}}$ (Constr. 2);

Table 2: BABE cut-and-choose setup cost for different parameter choices.

N_{CC}	M_{CC}	Setup time	Peak RAM usage (GB)	Garbling (s)	Evaluation (s)
78	10	0:06	1.27	1.44	1.45
95	9	0:06	1.21	1.85	1.32
124	8	0:07	1.29	2.54	1.18
181	7	0:09	1.05	3.57	1.05
307	6	0:14	1.12	6.32	0.86
669	5	0:28	1.46	13.22	0.74
2,268	4	1:37	1.83	49.15	0.59
18,756	3	12:31	3.93	383.87	0.41

N_{CC}: total cut-and-choose instances. M_{CC}: finalized instances for evaluation. Setup time is wall-clock time for cut-and-choose setup (timings exclude network latency). Setup time includes both Prover and Verifier computation. Garbling time is measured on the garbler node for all N_{CC} instances (not summed across both roles). Evaluation time is total decoding time for all finalized instances using the on-chain input labels (Sec. 5.6). Peak RAM usage is the maximum RAM used during execution. Hardware: CPU: AMD Ryzen 7 7840U(16 CPU);.

Parameter choices and statistical security. We choose the $(N_{\mathsf{CC}}, M_{\mathsf{CC}})$ pairs in Tab. 2 so that the soundness error of the cut-and-choose setup $\binom{N_{\mathsf{CC}}}{M_{\mathsf{CC}}}^{-1}$ is at most 2^{-40}. In this step, the Verifier opens $N_{\mathsf{CC}} - M_{\mathsf{CC}}$ randomly chosen instances and retains M_{CC} unopened instances as finalized. Soundness can fail only if all opened instances are correct while all M_{CC} finalized instances are incorrect.

Comparison to BitVM3. As an open-sourced reference point for garbled-circuit-based Groth16 verification, we additionally report cut-and-choose setup costs for BitVM3 [Bit25b]. To make the comparison parameter-aligned, we use the same $(N_{\mathsf{CC}}, M_{\mathsf{CC}})$ pairs as in Tab. 2 and report setup time and the garbling/evaluation breakdown. Other garbled-circuit-based approaches do not provide public implementations and benchmarks at comparable levels of detail. We use $(N_{\mathsf{CC}}, M_{\mathsf{CC}}) = (181, 7)$ as the main operating point for BitVM3, since the garbling time at this point is not very large while the on-chain footprint remains acceptable. We discuss augmenting BitVM3 with zk-SNARK-soldering in Sec. 9.3.

Operating points. For BitVM3, we primarily use $(N_{\mathsf{CC}}, M_{\mathsf{CC}}) = (181, 7)$; for BABE we use $M_{\mathsf{CC}} = 4$ in the end-to-end evaluation (cf. Tabs. 2 and 6).

On-chain implication. In our end-to-end on-chain evaluation we focus on $M_{\mathsf{CC}} = 4$. Cut-and-choose only guarantees that at least one of the 4 instances is correct with high probability but the Prover doesn't know which one. Hence in the baseline, four sets of input labels must be posted on chain. This directly drives the on-chain footprint and motivates soldering.

9.3 Soldering (zk-SNARK-soldering)

zk-SNARK-soldering is an optimization that reduces the number of distinct on-chain input-label sets from M_{CC} down to one, by binding finalized instances to a base instance and proving correctness of this binding

the evaluation component measures the Prover's evaluation of $\mathsf{ct}_{\mathsf{GC}}$ on the on-chain input labels (Sec. 5.6) to derive the proving ciphertext $\mathsf{ct}_{\mathsf{prove}}$ (Sec. 6.1.2).

Table 3: BitVM3 cut-and-choose setup cost for different parameter choices (same $(N_{\mathsf{CC}}, M_{\mathsf{CC}})$ grid as Tab. 2).

N_{CC}	M_{CC}	Setup time	Garbling	Evaluation
78	10	1:28:54	0:44:27	0:03:45
95	9	1:47:58	0:53:59	0:03:45
124	8	2:19:06	1:09:33	0:03:45
181	7	3:24:24	1:42:12	0:03:45
307	6	5:43:24	2:51:42	0:03:45
669	5	12:23:06	6:11:33	0:03:45
2,268	4	41:02:00	20:31:00	0:03:45

Times are in h:mm:ss format. Garbling time is from [Bit25b]; setup time is estimated as twice the garbling time. Evaluation time is expected to be negligible relative to garbling. Hardware: CPU: AMD Ryzen 7 7840U(16 CPU);.

with a soldering zk-SNARK proof. Our prototype uses the SP1 zkVM as the proving backend, but the stack is interchangeable and could be replaced by any zk-SNARK or zk-STARK system.

Soldering idea (informal). Let $\mathcal{I}$ denote the set of finalized instances and let $b = \min(\mathcal{I})$ be a base instance. Let $\{L_{i,j,0}, L_{i,j,1}\}_{j=1}^{2n}$ denote the garbler's per-wire label pairs for instance i ($2n$ is the number of input bits, with n as in Sec. 5). For each $i \in \mathcal{I}\setminus\{b\}$, define the per-wire, per-bit deltas $\Delta_{i,j,\beta} := L_{i,j,\beta} \oplus L_{b,j,\beta}$ for $\beta \in \{0,1\}, j \in [2n]$. The soldering zk-SNARK proof attests that these deltas are consistent with the commitments fixed during cut-and-choose, enabling derivation of all finalized per-wire label pairs from the base instance's labels. Equivalently, the proof certifies that there exists a single collection of per-wire label pairs for the base instance and for all other finalized instances, and that the deltas $\{\Delta_{i,j,\beta}\}$ are exactly the XOR differences between these labels. This check is performed by verifying a single soldering zk-SNARK proof, and any verification failure aborts the protocol.

Soldering reduces on-chain cost because only the base instance's tag set is posted on-chain. All other finalized instances' per-wire label pairs can be derived off-chain from the base labels and the proven deltas $\{\Delta_{i,j,\beta}\}$, so the on-chain footprint no longer scales with M_{CC}.

We focus on $M_{\mathsf{CC}} = 4$ because soldering introduces additional zk-SNARK proving time during setup. In particular, approaches that treat each finalized instance separately lead to overhead that grows with M_{CC}. We therefore focus on the $M_{\mathsf{CC}} = 4$ operating point in the end-to-end evaluation, and use $M_{\mathsf{CC}} = 5$ only as a reference point when discussing scaling.

We report soldering times for $M_{\mathsf{CC}} \in \{5, 6, 7\}$ (BitVM3) and $M_{\mathsf{CC}} \in \{4, 5, 6\}$ (BABE) in Tabs. 4 and 5, and compare the baseline and zk-SNARK-soldering configurations at $M_{\mathsf{CC}} = 4$ in Tab. 6.

BitVM3 with zk-SNARK-soldering. We can also augment BitVM3 with zk-SNARK-soldering to reduce the on-chain input-label footprint from M_{CC} sets to one. The soldering overhead depends on M_{CC} and the number of input labels, but is independent of the specific garbled circuit; thus the BABE soldering times in Tab. 5 serve as a lower bound, while the BitVM3 soldering times in Tab. 4 apply directly to BitVM3. Total setup time for BitVM3 with zk-SNARK-soldering is the sum of the C&C setup time from Tab. 3 and the corresponding soldering overhead.

Implementation. All reported measurements are obtained from our prototype implementation.

Table 4: BitVM3 soldering overhead (1019 labels) as a function of the number of finalized instances.

M_{CC}	Soldering time (s)
5	1529.11
6	1740.24
7	1977.10

Soldering time is the time to generate a zk-SNARK proof attesting to the correctness of input-label bindings across M_{CC} finalized instances. Hardware: CPU: AMD Ryzen 7 7840U(16 CPU);.

Table 5: BABE soldering overhead (508 labels) as a function of the number of finalized instances.

M_{CC}	Soldering time (s)
4	745.50
5	865.59
6	969.37

Soldering time is the time to generate a zk-SNARK proof attesting to the correctness of input-label bindings across M_{CC} finalized instances. Hardware: CPU: AMD Ryzen 7 7840U(16 CPU);.

Table 6: Off-chain timings at the operating point $M_{\mathsf{CC}} = 4$.

Configuration	Setup time for Verifier (s)	Evaluation time (s)
C&C baseline	49.15	0.59
C&C + zk-SNARK-soldering	49.15 + 745.50s	0.59

9.4 Verifiable Shamir Secret Sharing

C&C + VSSS (general idea). As a possible improvement over the C&C baseline, we consider compressing the on-chain input-label footprint using verifiable Shamir secret sharing (VSSS), inspired by Glock [Eag25] and the "efficient verifiable cut-and-choose" design notes [Lab25, BOB25b]. In the setup phase, the Verifier secret-shares the Prover input labels across instances and publishes commitments to the shares, while binding each instance to the committed values (e.g., via hashes and nonce commitments). In the proving phase, the Prover interpolates from the committed shares to reconstruct the input labels for the M_{CC} finalized instances and proceeds as in the baseline. Conceptually, this targets the same bottleneck as zk-SNARK-soldering—reducing the number of distinct on-chain input-label sets from M_{CC} down to one—but replaces zk proving with interpolation and commitment checks.

BitVM3 baseline at $(N_{\mathsf{CC}}, M_{\mathsf{CC}}) = (181, 7)$. We continue to use $(N_{\mathsf{CC}}, M_{\mathsf{CC}}) = (181, 7)$ as the main BitVM3 operating point (cf. Tab. 3). At this point, the reported BitVM3 setup time is `3:24:24` (with garbling time `1:42:12`).[19] To isolate what VSSS adds, we report below the incremental VSSS overhead as a function of $(N_{\mathsf{CC}}, M_{\mathsf{CC}})$.

[19]See the caption of Tab. 3 for how setup time is estimated from garbling time.

VSSS overhead across parameter sets. Tab. 7 reports the incremental overhead of adding the VSSS layer on top of cut-and-choose for different $(N_{\mathsf{CC}}, M_{\mathsf{CC}})$ choices.

These measurements come from a draft prototype intended to estimate the overhead under a mock-on-chain design (no end-to-end transaction integration). At the smallest operating point $(N_{\mathsf{CC}}, M_{\mathsf{CC}}) = (78, 10)$, the measured setup-time overhead is 32.96 s. Considering that N_{CC} garbling of BABE will take less than a second, we can estimate that the setup time will be around 40 seconds.

Table 7: Incremental overhead of VSSS over cut-and-choose for different parameter choices (prototype measurements).

N_{CC}	M_{CC}	Setup time overhead (s)	Peak RAM (GB)
78	10	32.96	0.98
95	9	40.86	1.20
124	8	53.84	1.57
181	7	83.93	2.31
307	6	159.23	3.92
669	5	332.61	8.56

Measurements run the VSSS layer standalone with the corresponding $(N_{\mathsf{CC}}, M_{\mathsf{CC}})$ parameters and 508 input bits. Setup time overhead is wall-clock time for setup between two roles (Prover and Verifier) executed on a single machine (timings exclude network latency). Hardware: CPU: AMD Ryzen 7 7840U(16 CPU);.

On-chain integration note. An end-to-end on-chain integration for VSSS still requires a concrete transaction design that ties the committed shares to the on-chain protocol logic (e.g., via adaptor signatures (ad-sig) and nonce commitments). As a result, we do not yet report on-chain costs for VSSS. Nevertheless, relative to prior BitVM3-centric experiments [BOB25b] we expect substantially lower on-chain footprint, since BABE requires far fewer evaluator input labels than BitVM3.

Acknowledgements

We thank the Babylon Labs team members for performing the on-chain experiments and for designing the illustrations. We thank Liam Eagen and Ying Tong Lai for early discussions about [EL26]. We thank Ertem Nusret Tas for reviewing the security proofs of the paper. The work of first and second authors is supported in part by the AFOSR Award FA9550-24-1-0156, and research grants from the Bakar Fund, the Stellar Development Foundation, Supra Inc., and Byzantine Research Inc. The work of the fifth author is partially supported by the Stanford Input-Output Global Research Hub. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these institutes.

References

[AAL+24] Lukas Aumayr, Zeta Avarikioti, Robin Linus, Matteo Maffei, Andrea Pelosi, Christos Stefo, and Alexei Zamyatin. BitVM: Quasi-turing complete computation on Bitcoin. Cryptology ePrint Archive, Report 2024/1995, 2024. 4

[AFP25] Amit Agarwal, Rex Fernando, and Benny Pinkas. Efficiently-thresholdizable batched identity based encryption, with applications. In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part III*, volume 16002 of *LNCS*, pages 69–100. Springer, Cham, August 2025. 8

[AIK04] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in NC^0. In *45th FOCS*, pages 166–175. IEEE Computer Society Press, October 2004. 9

[BBG05] Dan Boneh, Xavier Boyen, and Eu-Jin Goh. Hierarchical identity based encryption with constant size ciphertext. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 440–456. Springer, Berlin, Heidelberg, May 2005. 11

[BC16] Olivier Blazy and Céline Chevalier. Structure-preserving smooth projective hashing. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 339–369. Springer, Berlin, Heidelberg, December 2016. 7, 21

[BCMS20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Recursive proof composition from accumulation schemes. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 1–18. Springer, Cham, November 2020. 10

[BCPR14] Nir Bitansky, Ran Canetti, Omer Paneth, and Alon Rosen. On the existence of extractable one-way functions. In David B. Shmoys, editor, *46th ACM STOC*, pages 505–514. ACM Press, May / June 2014. 20

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 276–294. Springer, Berlin, Heidelberg, August 2014. 10

[BFOQ25] Jan Bormet, Sebastian Faust, Hussien Othman, and Ziyan Qu. BEAT-MEV: Epochless approach to batched threshold encryption for MEV prevention. In Lujo Bauer and Giancarlo Pellegrino, editors, *USENIX Security 2025*, pages 3457–3476. USENIX Association, August 2025. 8

[BHR12a] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Adaptively secure garbling with applications to one-time programs and secure outsourcing. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 134–153. Springer, Berlin, Heidelberg, December 2012. 15, 33

[BHR12b] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012. 14

[Bit25a] Bitlayer. Bitvm bridge testnet | bitlayer. https://docs.bitlayer.org/docs/BitVMBridge/Multi-Chain/testnet/, 2025. Last accessed: 2024-01-08'. 1

[Bit25b] BitVM. Garbled snark verifier. https://github.com/BitVM/garbled-snark-verifier, 2025. GitHub repository. Last accessed: 2026-01-11. 6, 46, 47, 48

[BL20] Fabrice Benhamouda and Huijia Lin. Mr NISC: Multiparty reusable non-interactive secure computation. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 349–378. Springer, Cham, November 2020. 8

[BMM$^+$21] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. Proofs for inner pairing products and applications. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 65–97. Springer, Cham, December 2021. 10

[BN06] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. In Bart Preneel and Stafford Tavares, editors, *SAC 2005*, volume 3897 of *LNCS*, pages 319–331. Springer, Berlin, Heidelberg, August 2006. 6

[BOB25a] BOB. Bob | bridge. https://app.gobob.xyz/en/bridge, 2025. Last accessed: 2024-01-08'. 1

[BOB25b] BOB. BOB lowers onchain costs for BitVM3 via cut-and-choose implementation to $10.91. https://gobob.xyz/blog/bob-lowers-onchain-costs-for-bitvm3, December 2025. BOB Blog. Last accessed: 2026-01-12. 49, 50

[Boy08] Xavier Boyen. The uber-assumption family (invited talk). In Steven D. Galbraith and Kenneth G. Paterson, editors, *PAIRING 2008*, volume 5209 of *LNCS*, pages 39–56. Springer, Berlin, Heidelberg, September 2008. 11

[BP15] Elette Boyle and Rafael Pass. Limits of extractability assumptions with distributional auxiliary input. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 236–261. Springer, Berlin, Heidelberg, November / December 2015. 20

[Cat25] Catalyst. Bitcoin prism. https://github.com/catalystsystem/bitcoinprism-evm, 2025. GitHub repository. Last accessed: 2026-01-14. 4

[CFH$^+$22] Matteo Campanelli, Dario Fiore, Semin Han, Jihye Kim, Dimitris Kolonelos, and Hyunok Oh. Succinct zero-knowledge batch proofs for set accumulators. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 455–469. ACM Press, November 2022. 10

[CFK24] Matteo Campanelli, Dario Fiore, and Hamidreza Khoshakhlagh. Witness encryption for succinct functional commitments and applications. In Qiang Tang and Vanessa Teague, editors, *PKC 2024, Part II*, volume 14602 of *LNCS*, pages 132–167. Springer, Cham, April 2024. 8

[CGPW25] Arka Rai Choudhuri, Sanjam Garg, Guru-Vamsi Policharla, and Mingyuan Wang. Practical mempool privacy via one-time setup batched threshold encryption. In Lujo Bauer and Giancarlo Pellegrino, editors, *USENIX Security 2025*, pages 3477–3495. USENIX Association, August 2025. 8

[Che25] Weikeng Chen. SoK: BitVM with succinct on-chain cost. Cryptology ePrint Archive, Report 2025/1253, 2025. 5, 39

[Cit25] Citrea. Citrea bridge | citrea. https://citrea.xyz/bridge, 2025. Last accessed: 2024-01-08'. 1

[DLT$^+$24] Xinshu Dong, Orfeas Stefanos Thyfronitis Litos, Ertem Nusret Tas, David Tse, Robin Linus Woll, Lei Yang, and Mingchao Yu. Remote staking with economic safety. *CoRR*, abs/2408.01896, 2024. 5

[Dry17] Thaddeus Dryja. Discreet log contracts. https://static1.squarespace.com/static/6675a0d5fc9e317c60db9b37/t/66e4597b7f23866561a64a95/1726241147904/discreet+log+contracts+paper.pdf, 2017. Last accessed: 2026-01-12. 6

[Eag25] Liam Eagen. Glock: Garbled locks for Bitcoin. Cryptology ePrint Archive, Report 2025/1485, 2025. 5, 6, 49

[EL26] Liam Eagen and Ying Tong Lai. Argo MAC: Garbling with elliptic curve MACs. Cryptology ePrint Archive, Paper 2026/049, 2026. 1, 9, 23, 50

[FBFL25] Ariel Futoransky, Fadi Barbàra, Ramses Fernandez, and Gabriel Larotonda. OHMG: One hot modular garbling. Cryptology ePrint Archive, Report 2025/2338, 2025. 6

[FHAS24] Nils Fleischhacker, Mathias Hall-Andersen, and Mark Simkin. Extractable witness encryption for KZG commitments and efficient laconic OT. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part II*, volume 15485 of *LNCS*, pages 423–453. Springer, Singapore, December 2024. 8

[FKdP23] Dario Fiore, Dimitris Kolonelos, and Paola de Perthuis. Cuckoo commitments: Registration-based encryption and key-value map commitments for large spaces. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part V*, volume 14442 of *LNCS*, pages 166–200. Springer, Singapore, December 2023. 8

[FKN94] Uri Feige, Joe Kilian, and Moni Naor. A minimal model for secure computation. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, STOC '94, pages 554–563, New York, NY, USA, 1994. ACM. 9, 29

[GGKS25] Sanjam Garg, Aarushi Goel, Dimitris Kolonelos, and Rohit Sinha. Jigsaw: Doubly private smart contracts. Cryptology ePrint Archive, Report 2025/1147, 2025. 10

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Berlin, Heidelberg, May 2013. 12

[GGSW13] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th ACM STOC*, pages 467–476. ACM Press, June 2013. 6, 14

[GGW24] Sanjam Garg, Aarushi Goel, and Mingyuan Wang. How to prove statements obliviously? In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part X*, volume 14929 of *LNCS*, pages 449–487. Springer, Cham, August 2024. 10

[GHK$^+$25] Sanjam Garg, Mohammad Hajiabadi, Dimitris Kolonelos, Abhiram Kothapalli, and Guru-Vamsi Policharla. A framework for witness encryption from linearly verifiable SNARKs and applications. In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part III*, volume 16002 of *LNCS*, pages 504–539. Springer, Cham, August 2025. 8, 21

[GIKM00] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. *J. Comput. Syst. Sci.*, 60(3):592–629, 2000. 5

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, Berlin, Heidelberg, April 2015. 15, 18

[GKP$^+$13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run Turing machines on encrypted data. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 536–553. Springer, Berlin, Heidelberg, August 2013. 6, 13

[GKPW24] Sanjam Garg, Dimitris Kolonelos, Guru-Vamsi Policharla, and Mingyuan Wang. Threshold encryption with silent setup. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part VII*, volume 14926 of *LNCS*, pages 352–386. Springer, Cham, August 2024. 1, 7, 8, 21

[GMN22] Nicolas Gailly, Mary Maller, and Anca Nitulescu. SnarkPack: Practical SNARK aggregation. In Ittay Eyal and Juan A. Garay, editors, *FC 2022*, volume 13411 of *LNCS*, pages 203–229. Springer, Cham, May 2022. 10

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Berlin, Heidelberg, May 2016. 6, 12, 13

[Her18] Maurice Herlihy. Atomic cross-chain swaps. *arXiv preprint arXiv:1801.09515*, 2018. Last accessed: 2026-01-12. 5

[Hio22] Leona Hioki. Trustless bitcoin bridge creation with witness encryption. https://ethresear.ch/t/trustless-bitcoin-bridge-creation-with-witness-encryption/11953, February 2022. Ethereum Research. Last accessed: 2026-01-11. 7

[IK00] Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *41st FOCS*, pages 294–304. IEEE Computer Society Press, November 2000. 9

[IK02] Yuval Ishai and Eyal Kushilevitz. Perfect constant-round secure computation via perfect randomizing polynomials. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors, *ICALP 2002*, volume 2380 of *LNCS*, pages 244–256. Springer, Berlin, Heidelberg, July 2002. 9

[Ish13] Yuval Ishai. Randomization techniques for secure computation. In Manoj Prabhakaran and Amit Sahai, editors, *Secure Multi-Party Computation*, volume 10 of *Cryptology and Information Security Series*, pages 222–248. IOS Press, 2013. 9, 23, 28, 29

[IW14] Yuval Ishai and Hoeteck Wee. Partial garbling schemes and their applications. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *ICALP 2014, Part I*, volume 8572 of *LNCS*, pages 650–662. Springer, Berlin, Heidelberg, July 2014. 9

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 486–498. Springer, Berlin, Heidelberg, July 2008. 6, 15

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 359–388. Springer, Cham, August 2022. 10

[LAA$^+$25] Robin Linus, Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, Andrea Pelosi, Orfeas Thyfronitis Litos, Christos Stefo, David Tse, and Alexei Zamyatin. Bridging Bitcoin to second layers via BitVM2. Cryptology ePrint Archive, Report 2025/1158, 2025. To appear in Usenix Security 2026. 1, 4, 5, 6, 18, 45

[Lab25] Alpen Labs. Efficient verifiable cut and choose for glock. https://hackmd.io/@alpen/B1QfSSO5gg, 2025. HackMD. Last accessed: 2026-01-12. 49

[Lam79] Leslie Lamport. Constructing digital signatures from a one-way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, October 1979. 5, 15, 16

[Lin23] Robin Linus. Bitvm: Compute anything on bitcoin. https://bitvm.org/bitvm.pdf, December 2023. Last accessed: 2024-01-08'. 4

[Lin24] Robin Linus. Bitvm 3s - garbled circuits for efficient computation on bitcoin. https://bitvm.org/bitvm3.pdf, 2024. Last accessed: 2024-01-08'. 1, 5

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 52–78. Springer, Berlin, Heidelberg, May 2007. 6

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009. 14, 33

[Mau05] Ueli M. Maurer. Abstract models of computation in cryptography (invited paper). In Nigel P. Smart, editor, *10th IMA International Conference on Cryptography and Coding*, volume 3796 of *LNCS*, pages 1–12. Springer, Berlin, Heidelberg, December 2005. 10, 11

[MLLP25] Varun Madathil, Arthur Lazzaretti, Zeyu Liu, and Charalampos Papamanthou. TACITA: Threshold aggregation without client interaction. Cryptology ePrint Archive, Report 2025/1579, 2025. 8

[Nak09] Satoshi Nakamoto. Bitcoin open source implementation of p2p currency. https://satoshi.nakamotoinstitute.org/posts/p2pfoundation/1/, 2009. Last accessed: 2024-01-08'. 4

[OKMZ25] Michele Orrù, George Kadianakis, Mary Maller, and Greg Zaverucha. Beyond the circuit: How to minimize foreign arithmetic in ZKP circuits. *CiC*, 2(1):23, 2025. 10

[OWWB20] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020*, pages 2075–2092. USENIX Association, August 2020. 10

[PD16] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. https://lightning.network/lightning-network-paper.pdf, 2016. Last accessed: 2026-01-12. 5

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society Press, May 2013. 12

[Rub24] Jeremy Rubin. Delbrag. https://rubin.io/public/pdfs/delbrag.pdf, 2024. Last accessed: 2024-01-08'. 1, 5

[SGB24] István András Seres, Noemi Glaeser, and Joseph Bonneau. Short paper: Naysayer proofs. In Jeremy Clark and Elaine Shi, editors, *FC 2024, Part II*, volume 14745 of *LNCS*, pages 22–32. Springer, Cham, March 2024. 5

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Berlin, Heidelberg, May 1997. 10

[Wik20] Bitcoin Wiki. Atomic swap. https://en.bitcoin.it/wiki/Atomic_swap, 2020. Last accessed: 2026-01-12. 5

[Wik21] Bitcoin Wiki. Hash time locked contracts. https://en.bitcoin.it/wiki/Hash_Time_Locked_Contracts, 2021. Last accessed: 2026-01-12. 5

[WNT20] Pieter Wuille, Jonas Nick, and Anthony Towns. Bip 0341, taproot: Segwit version 1 spending rules. https://en.bitcoin.it/wiki/BIP_0341, January 2020. Last accessed: 2024-01-08'. 15

[WOS+25] Anna P. Y. Woo, Alex Ozdemir, Chad Sharp, Thomas Pornin, and Paul Grubbs. Efficient proofs of possession for legacy signatures. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *2025 IEEE Symposium on Security and Privacy*, pages 3291–3308. IEEE Computer Society Press, May 2025. 10

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982. 5, 9, 14

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Berlin, Heidelberg, April 2015. 6, 15

A Honest-Setup BABE Protocol Details

A.1 Transactions

The detailed specifications of the transactions are given in this section. Each transaction specifies the inputs, outputs, the locking scripts for each input and output, and the transaction witnesses.

A.1.1 Notation

We follow the transaction model of Sec. 2.6: a transaction is $\mathsf{tx} = (\mathsf{inputs}, \mathsf{tx\_witnesses}, \mathsf{outputs})$. Each input is of the form $\mathsf{in} = (\mathsf{PrevTx}, \mathsf{outIndex}, \mathsf{leaf})$, where PrevTx is the previous transaction, $\mathsf{outIndex}$ is the index of the output in that transaction, and leaf is the leaf of the Taproot tree to be satisfied for that input. Each output is $\mathsf{out} = (a, \mathsf{lockScript})$ where a is the value in that UTXO and $\mathsf{lockScript}$ is the locking script. The i-th witness in $\mathsf{tx\_witnesses}$ is the data (e.g., signatures, hash preimages) supplied to satisfy the i-th input's leaf script.

In the tables below, $*$ denotes that the corresponding field can be *any* value of the appropriate type. For example, in an input $(*, *, *)$, the first component can be any previous transaction, the second any output index, and the third any leaf (i.e., any UTXO and any leaf thereof). When a component is not $*$, it is fixed: e.g., $(\mathsf{tx}_{\mathsf{Assert}}, 0, \mathsf{leaf})$ means the input references the 0-th output of the Assert transaction and the spender must satisfy the leaf leaf.

Output locking scripts are often written as $\langle \mathsf{leaf}_0, \ldots, \mathsf{leaf}_{k-1} \rangle$. This denotes a *Taproot tree* (taptree): the UTXO can be spent by satisfying *one* of the leaves leaf_j. A transaction that spends such an output must specify which leaf it is satisfying and supply a witness that satisfies that leaf's script. The locking scripts $\mathsf{CheckSig}$, $\mathsf{RelTimelock}$, $\mathsf{HashLock}$, $\mathsf{CheckLampSig}$, $\mathsf{CheckLampSigsMatch}$, and their combinations are as defined in Sec. 2.6.

A.1.2 Example: The Assert Transaction

We illustrate the notation and the role of inputs, outputs, and witnesses using the Assert transaction as an example.

Inputs and input script. The Assert transaction has a single input: $(*, *, \langle \mathsf{CheckLampSig}(\mathsf{lpk}_P)\rangle)$. The first two components are $*$: the input may reference *any* UTXO (any previous transaction and any output index). The third component fixes the *leaf* to be satisfied: $\langle \mathsf{CheckLampSig}(\mathsf{lpk}_P)\rangle$. So the UTXO being spent must have a locking script that is a Taproot tree containing the leaf $\mathsf{CheckLampSig}(\mathsf{lpk}_P)$ (e.g., a tree that includes this leaf among others). In practice, this is typically another UTXO set up so that its taptree includes this leaf. To spend it, the Prover must provide a witness that satisfies $\mathsf{CheckLampSig}(\mathsf{lpk}_P)$, i.e., a Lamport signature under lpk_P on some message (see Eq. (15)).

Witness. The Tx Witness for input (0) is $\mu_1, \ldots, \mu_{2n}$. These are the $2n$ components of the Lamport signature μ on the message (e.g., the proof element π_1; n is the bit length of $\mathbb{F}_p$ as in Sec. 5): for each bit of the message, the signer reveals the corresponding preimage from the Lamport key. This witness satisfies $\mathsf{CheckLampSig}(\mathsf{lpk}_P)$ and thus binds the Prover to a single message when posting the Assert transaction.

Outputs and their scripts. The Assert transaction has two outputs, both with amount 0.

- **Output (0)** has locking script $\langle \mathsf{RelTimelock}(\Delta_2) \wedge \mathsf{CheckSig}(\mathsf{pk}_P) \wedge \mathsf{CheckSig}(\mathsf{pk}_V),\ \mathsf{CheckSig}(\mathsf{pk}_P) \wedge \mathsf{CheckSig}(\mathsf{pk}_V)\rangle$. This is a Taproot tree with *two* leaves. The UTXO can be spent by satisfying *exactly one* of them:

  1. **First leaf** $\mathsf{RelTimelock}(\Delta_2) \wedge \mathsf{CheckSig}(\mathsf{pk}_P) \wedge \mathsf{CheckSig}(\mathsf{pk}_V)$: the spending transaction must be included at least Δ_2 blocks after the Assert transaction, and the witness must contain signatures σ_P, σ_V on the transaction skeleton under pk_P and pk_V. This path is used by the $\mathsf{Withdraw}$ transaction.
  2. **Second leaf** $\mathsf{CheckSig}(\mathsf{pk}_P) \wedge \mathsf{CheckSig}(\mathsf{pk}_V)$: the witness must contain σ_P and σ_V (no time-lock). This path is used by the $\mathsf{NoWithdraw}$ transaction.

  Therefore, output (0) of Assert can be spent either (i) after Δ_2 blocks with both parties' Schnorr signatures, or (ii) immediately with both parties' Schnorr signatures.

- **Output (1)** has locking script $\langle \mathsf{CheckLampSigsMatch}(\mathsf{lpk}_P, \mathsf{lpk}_V) \wedge \mathsf{CheckSig}(\mathsf{pk}_V) \wedge \mathsf{CheckSig}(\mathsf{pk}_P)\rangle$. There is a single leaf, so this output can be spent only one way: the witness must satisfy $\mathsf{CheckLampSigsMatch}(\mathsf{lpk}_P, \mathsf{l}$
  (Lamport signatures under lpk_P and lpk_V on the *same* message; see Eq. (16)) and provide σ_V, σ_P. This path is used by the ChallengeAssert transaction.

Deposit **Transaction**	
Inputs	(0) $(*, *, *)$
Outputs	(0) $(v, \langle \mathsf{CheckSig}(\mathsf{pk}_P) \wedge \mathsf{CheckSig}(\mathsf{pk}_V)\rangle)$
Tx Witness	(0) $*$

Assert **Transaction**	
Inputs	(0) $(*, *, \langle \mathsf{CheckLampSig}(\mathsf{lpk}_P)\rangle)$ (see Eq. (15))
Outputs	(0) $(0, \langle \mathsf{RelTimelock}(\Delta_2) \wedge \mathsf{CheckSig}(\mathsf{pk}_P) \wedge \mathsf{CheckSig}(\mathsf{pk}_V), \mathsf{CheckSig}(\mathsf{pk}_P) \wedge \mathsf{CheckSig}(\mathsf{pk}_V)\rangle)$ (1) $(0, \langle \mathsf{CheckLampSigsMatch}(\mathsf{lpk}_P, \mathsf{lpk}_V) \wedge \mathsf{CheckSig}(\mathsf{pk}_V) \wedge \mathsf{CheckSig}(\mathsf{pk}_P)\rangle)$ (see Eq. (16))
Tx Witness	(0) $\mu_1, \ldots, \mu_\ell$

ChallengeAssert **Transaction**	
Inputs	(0) $(\mathsf{tx}_{\mathsf{Assert}}, 1, \langle \mathsf{CheckLampSigsMatch}(\mathsf{lpk}_P, \mathsf{lpk}_V) \wedge \mathsf{CheckSig}(\mathsf{pk}_V) \wedge \mathsf{CheckSig}(\mathsf{pk}_P)\rangle)$ (see Eq. (16))
Outputs	(0) $(0, \langle \mathsf{RelTimelock}(\Delta_1) \wedge \mathsf{CheckSig}(\mathsf{pk}_V), \mathsf{HashLock}(\mathsf{Hash}_{BTC}(\mathsf{msg})) \wedge \mathsf{CheckSig}(\mathsf{pk}_P)\rangle)$
Tx Witness	(0) $L_1, \ldots, L_\ell, \mu_1, \ldots, \mu_\ell, \sigma_V, \sigma_P$

NoWithdraw **Transaction**	
Inputs	(0) $(\mathsf{tx}_{\mathsf{Assert}}, 0, \mathsf{CheckSig}(\mathsf{pk}_P) \wedge \mathsf{CheckSig}(\mathsf{pk}_V))$ (1) $(\mathsf{tx}_{\mathsf{ChallengeAssert}}, 0, \mathsf{RelTimelock}(\Delta_1) \wedge \mathsf{CheckSig}(\mathsf{pk}_V))$
Outputs	(0) $(0, \langle \mathsf{CheckSig}(\mathsf{pk}_V)\rangle)$
Tx Witness	(0) σ_P, σ_V
	(1) σ_V

WronglyChallenged **Transaction**	
Inputs	(0) $(\mathsf{tx}_{\mathsf{ChallengeAssert}}, 0, \mathsf{HashLock}(h_{\mathsf{msg}}) \wedge \mathsf{CheckSig}(\mathsf{pk}_P))$
Outputs	(0) $(0, \langle \mathsf{CheckSig}(\mathsf{pk}_P)\rangle)$
Tx Witness	(0) σ_P, msg

Withdraw **Transaction**	
Inputs	(0) $(\mathsf{tx}_{\mathsf{Deposit}}, 0, \mathsf{CheckSig}(\mathsf{pk}_P) \wedge \mathsf{CheckSig}(\mathsf{pk}_V))$ (1) $(\mathsf{tx}_{\mathsf{Assert}}, 0, \mathsf{RelTimelock}(\Delta_2) \wedge \mathsf{CheckSig}(\mathsf{pk}_P) \wedge \mathsf{CheckSig}(\mathsf{pk}_V))$
Outputs	(0) $(v, \langle \mathsf{CheckSig}(\mathsf{pk}_P)\rangle)$
Tx Witness	(0) σ_P, σ_V
	(1) σ_P, σ_V

B Protocol for Malicious Security

The setup protocol using cut-and-choose, which achieves the BitVM-core properties in Defs. 9 to 11, is shown in Algs. 4 and 5.

The corresponding proving phase is shown in Alg. 6.

Algorithm 4 $\mathtt{Setup}$ algorithms for malicious security (Prover)

```
1:  procedure $P_{\mathtt{Setup,mal}}(\mathsf{crs})$                                          ▷ Run by Prover
2:      $(\mathsf{sk}_P, \mathsf{pk}_P) \leftarrow \mathsf{Sig}_{BTC}.\mathtt{Gen}(1^\lambda)$                ▷ Sample signing key
3:      send $(\mathsf{pk}_P)$ to Verifier

        Upon receiving $\left(\mathsf{pk}_V, \{h_{\mathsf{msg},i}, \mathsf{epk}_i, h_{\mathsf{ct_{setup}},i}\}_{i=1}^{N_{\mathsf{CC}}}\right)$ from Verifier:
4:      Sample $\mathcal{I}$ as a uniformly random subset of $[N_{\mathsf{CC}}]$ of size $M_{\mathsf{CC}}$
5:      send $\mathcal{I}$ to Verifier

        Upon receiving $\left(\{\mathsf{msg}_i, r_i, \mathsf{seed}_i\}_{i\in[N_{\mathsf{CC}}]\setminus\mathcal{I}}, \{\mathsf{ct_{setup}}_i, \mathsf{ct_{GC}}_i, \mathsf{ek}_i\}_{i\in\mathcal{I}}\right)$ from Verifier:
6:      $\boldsymbol{x} \leftarrow \mathsf{GenStmt}(\mathsf{pk}_P)$
7:      for $i \in [N_{\mathsf{CC}}] \setminus \mathcal{I}$ do                             ▷ If any verification fails, abort
8:          Verify $\mathsf{WE}.\mathtt{Enc}_{\mathsf{setup}}(\mathsf{crs}, \boldsymbol{x}, \mathsf{msg}_i, r_i) = \mathsf{ct_{setup}}_i$              ▷ Constr. 1
9:          Verify $(\mathsf{ct_{GC}}_i, \mathsf{ek}_i) = \mathsf{Garble}(r_i; \mathsf{seed}_i)$                    ▷ Sec. 5.6
10:         Verify $h_{\mathsf{msg},i} = \mathsf{Hash}_{BTC}(\mathsf{msg}_i)$
11:         for $j \in \{1, \ldots, m\}, b \in \{0,1\}$ do
12:             Verify $((\mathsf{epk}_i)_j^b = \mathsf{Hash}_{BTC}(\mathsf{ek}_i)_j^b)$
13:         end for
14:         Verify $h_{\mathsf{ct_{setup}},i} = \mathsf{RO}(\mathsf{ct_{setup}}_i)$
15:     end for
16:     $(\mathsf{lsk}_P, \mathsf{lpk}_P) \leftarrow \mathsf{LampSig}.\mathtt{Gen}(1^\lambda)$                  ▷ Sample Lamport key
17:     $(\mathcal{T}, \mathcal{S}) \leftarrow \mathsf{CreateTxSetMalicious}\left(\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{lpk}_P, \{\mathsf{epk}_i\}_{i\in\mathcal{I}}, \{h_{\mathsf{msg},i}\}_{i\in\mathcal{I}}\right)$   ▷ Alg. 7
18:     $\mathsf{presigs}_P \leftarrow \mathsf{SignTxs}_P(\mathsf{sk}_P, \mathcal{T})$                          ▷ Alg. 3
19:     send $(\mathsf{pk}_P, \mathsf{lpk}_P, \mathsf{presigs}_P)$ to Verifier

        Upon receiving $(\mathsf{presigs}_V)$ from Verifier:
20:     $\mathsf{VerifySigs}_P(\mathsf{pk}_V, \mathcal{T}, \mathsf{presigs}_V)$                  ▷ Alg. 3; if fails, abort
21:     Sign $\mathsf{tx_{Deposit}}$ and submit to Bitcoin via $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx_{Deposit}})$
22:     $\mathsf{st}_P \leftarrow (\mathsf{sk}_P, \mathsf{lsk}_P, \mathsf{presigs}_V, \{\mathsf{ct_{setup}}_i, \mathsf{ct_{GC}}_i\}_{i\in\mathcal{I}})$
23:     return $(\boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P)$
24: end procedure
```

Algorithm 5 Setup algorithms for malicious security (Verifier)

```
1:  procedure $V_{\mathsf{Setup,mal}}(\mathsf{crs})$                                ▷ Run by Verifier
        Upon receiving $(\mathsf{pk}_P)$ from Prover:
2:      $\boldsymbol{x} \leftarrow \mathsf{GenStmt}(\mathsf{pk}_P)$          ▷ Application-specific: map Prover to statement
3:      $(\mathsf{sk}_V, \mathsf{pk}_V) \leftarrow \mathsf{Sig}_{BTC}.\mathtt{Gen}(1^\lambda)$          ▷ Sample signing key
4:      for $i = 1, \dots, N_{\mathsf{CC}}$ do          ▷ Generate ciphertexts for cut-and-choose
5:          $\mathsf{msg}_i \leftarrow\$ \mathcal{M}$, $r_i \leftarrow\$ \mathbb{F}_q^*$          ▷ Sample secrets
6:          $\mathsf{ct}_{\mathsf{setup}_i} \leftarrow \mathsf{WE}.\mathtt{Enc}_{\mathsf{setup}}(\mathsf{crs}, \boldsymbol{x}, \mathsf{msg}_i, r_i)$          ▷ WE ciphertext (Constr. 1)
7:          $\mathsf{seed}_i \leftarrow\$ \{0,1\}^\lambda$          ▷ Seed for Garble
8:          $\mathsf{ct}_{\mathsf{GC}i}, \mathsf{ek}_i \leftarrow \mathsf{Garble}(r_i; \mathsf{seed}_i)$          ▷ GC ciphertext and encoding key (Sec. 5.6)
9:          $h_{\mathsf{msg},i} \leftarrow \mathsf{Hash}_{BTC}(\mathsf{msg}_i)$          ▷ Hash message for hashlock
10:         for $j \in \{1, \dots, m\}, b \in \{0,1\}$ do
11:             $(\mathsf{epk}_i)_j^b \leftarrow \mathsf{Hash}_{BTC}((\mathsf{ek}_i)_j^b)$          ▷ Hash input labels for hashlock
12:         end for
13:         $h_{\mathsf{ct}_{\mathsf{setup}},i} \leftarrow \mathsf{RO}(\mathsf{ct}_{\mathsf{setup}_i})$
14:     end for
15:     send $\left(\mathsf{pk}_V, \{h_{\mathsf{msg},i}, \mathsf{epk}_i, h_{\mathsf{ct}_{\mathsf{setup}},i}\}_{i=1}^{N_{\mathsf{CC}}}\right)$ to Prover

        Upon receiving $\mathcal{I}$ from Prover:
16:     send $\left(\{\mathsf{msg}_i, r_i, \mathsf{seed}_i\}_{i \in [N_{\mathsf{CC}}] \setminus \mathcal{I}}, \{\mathsf{ct}_{\mathsf{setup}_i}, \mathsf{ct}_{\mathsf{GC}i}, \mathsf{ek}_i\}_{i \in \mathcal{I}}\right)$ to Prover

        Upon receiving $(\mathsf{pk}_P, \mathsf{lpk}_P, \mathsf{presigs}_P)$ from Prover:
17:     $\mathsf{VerifySigs}_V(\mathsf{pk}_P, \mathcal{T}, \mathsf{presigs}_P)$          ▷ Alg. 3; if fails, abort
18:     $(\mathcal{T}, \mathcal{S}) \leftarrow \mathsf{CreateTxSetMalicious}\left(\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{lpk}_P, \{\mathsf{epk}_i\}_{i \in \mathcal{I}}, \{h_{\mathsf{msg},i}\}_{i \in \mathcal{I}}\right)$          ▷ Alg. 7
19:     $\mathsf{presigs}_V \leftarrow \mathsf{SignTxs}_V(\mathsf{sk}_V, \mathcal{T})$          ▷ Alg. 3
20:     $\mathsf{st}_V \leftarrow (\mathsf{sk}_V, \{\mathsf{ek}_i\}_{i \in \mathcal{I}}, \mathsf{presigs}_P)$
21:     send $(\mathsf{presigs}_V)$ to Prover
22:     return $(\boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_V)$
23: end procedure
```

Algorithm 6 `Prove` algorithms

```
1:  procedure $P_{\texttt{Prove},\mathsf{mal}}(\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_P, \boldsymbol{w})$    ▷ Run by Prover
2:      Parse $\mathsf{st}_P = (\mathsf{sk}_P, \mathsf{lsk}_P, \mathsf{presigs}_V, \{\mathsf{ct}_{\mathsf{setup}_i}, \mathsf{ct}_{\mathsf{GC}\,i}\}_{i=1}^{M_{\mathsf{CC}}})$ and $\mathsf{presigs}_V = (\sigma^V_{\mathsf{Withdraw}})$
3:      $(\pi_1, \pi_2, \pi_3) \leftarrow \mathsf{Groth16}.\texttt{Prove}(\mathsf{crs}, \boldsymbol{x}, \boldsymbol{w})$
4:      $w_{\mathsf{Assert}} \leftarrow \mathsf{LampSig.Sign}(\mathsf{lsk}_P, \pi_1)$    ▷ Compute Lamport signature
5:      Post $\mathsf{tx}_{\mathsf{Assert}}$ with transaction witness $w_{\mathsf{Assert}}$: call $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{Assert}})$

        Upon seeing $\mathsf{tx}_{\mathsf{Assert}}$ and $\Delta_2$ new blocks after $\mathsf{tx}_{\mathsf{Assert}}$ in $\mathcal{L}_P$:
6:      $w_{\mathsf{Withdraw}} \leftarrow (\sigma^P_{\mathsf{Withdraw}}, \sigma^V_{\mathsf{Withdraw}})$ where $\sigma^P_{\mathsf{Withdraw}} \leftarrow \mathsf{Sig}_{BTC}.\mathsf{Sign}(\mathsf{sk}_P, \overline{\mathsf{tx}}_{\mathsf{Withdraw}})$
7:      Post $\mathsf{tx}_{\mathsf{Withdraw}}$ with transaction witness $w_{\mathsf{Withdraw}}$: call $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{Withdraw}})$
8:      return 1

        Upon seeing $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ in $\mathcal{L}_P$:
9:      Extract input labels $\{L_i\}_{i=1}^{M_{\mathsf{CC}}}$ from the transaction witness of $\mathsf{tx}_{\mathsf{ChallengeAssert}}$
10:     For $i = 1, \ldots, M_{\mathsf{CC}}$: $\mathsf{ct}_{\mathsf{prove}_i} \leftarrow \mathsf{EvalGC}(\mathsf{ct}_{\mathsf{GC}\,i}, L_i)$    ▷ Evaluate garbled circuits (Sec. 5.6)
11:     For $i = 1, \ldots, M_{\mathsf{CC}}$: $\mathsf{msg}_i \leftarrow \mathsf{WE.Dec}(\mathsf{ct}_{\mathsf{setup}_i}, \mathsf{ct}_{\mathsf{prove}_i}, \pi_2, \pi_3)$    ▷ Decrypt secrets (Constr. 1)
12:     Find $i$ such that $w_{\mathsf{WronglyChallenged}} \leftarrow (\sigma^P_{\mathsf{WronglyChallenged}}, \mathsf{msg}_i)$ where $\sigma^P_{\mathsf{WronglyChallenged}} \leftarrow$
    $\mathsf{Sig}_{BTC}.\mathsf{Sign}(\mathsf{sk}_P, \overline{\mathsf{tx}}_{\mathsf{WronglyChallenged}})$ is a valid transaction witness
13:     Post $\mathsf{tx}_{\mathsf{WronglyChallenged}}$ with transaction witness $w_{\mathsf{WronglyChallenged}}$: call $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{WronglyChallenged}})$
14: end procedure

15: procedure $V_{\texttt{Prove},\mathsf{mal}}(\mathsf{crs}, \boldsymbol{x}, \mathcal{T}, \mathcal{S}, \mathsf{st}_V)$    ▷ Run by Verifier
        Upon seeing $\mathsf{tx}_{\mathsf{Assert}}$ in $\mathcal{L}_V$:
16:     Parse $\mathsf{st}_V = (\mathsf{sk}_V, \{\mathsf{ek}_i\}_{i=1}^{M_{\mathsf{CC}}}, \mathsf{presigs}_P)$ and $\mathsf{presigs}_P = (\sigma^P_{\mathsf{ChallengeAssert}}, \sigma^P_{\mathsf{NoWithdraw}})$
17:     Extract $\pi_1$ and Lamport signature $\mu$ from the transaction witness of $\mathsf{tx}_{\mathsf{Assert}}$
18:     For $i = 1, \ldots, M_{\mathsf{CC}}$: $L_i \leftarrow \mathsf{Encode}(\mathsf{ek}_i, \pi_1)$    ▷ Compute input labels (Sec. 5.6)
19:     $w_{\mathsf{ChallengeAssert}} \leftarrow (\sigma^P_{\mathsf{ChallengeAssert}}, \sigma^V_{\mathsf{ChallengeAssert}}, \mu, \{L_i\}_{i=1}^{M_{\mathsf{CC}}})$ where $\sigma^V_{\mathsf{ChallengeAssert}} \leftarrow \mathsf{Sig}_{BTC}.\mathsf{Sign}(\mathsf{sk}_V, \overline{\mathsf{tx}}_{\mathsf{ChallengeAssert}})$
20:     Post $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ with transaction witness $w_{\mathsf{ChallengeAssert}}$: call $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{ChallengeAssert}})$

        Upon seeing $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ and $\Delta_1$ new blocks after $\mathsf{tx}_{\mathsf{ChallengeAssert}}$ in $\mathcal{L}_V$:
21:     $w_{\mathsf{NoWithdraw}} \leftarrow (\sigma^P_{\mathsf{NoWithdraw}}, \sigma^V_{\mathsf{NoWithdraw}})$ where $\sigma^V_{\mathsf{NoWithdraw}} \leftarrow \mathsf{Sig}_{BTC}.\mathsf{Sign}(\mathsf{sk}_V, \overline{\mathsf{tx}}_{\mathsf{NoWithdraw}})$
22:     Post $\mathsf{tx}_{\mathsf{NoWithdraw}}$ with transaction witness $w_{\mathsf{NoWithdraw}}$: call $\mathcal{F}_{\mathsf{BTC}}.\mathsf{WRITE}(\mathsf{tx}_{\mathsf{NoWithdraw}})$
23:     return 1
24: end procedure
```

Algorithm 7 Locking scripts for the protocol with malicious security (Algs. 4 and 5)

```
1: function CreateTxSetMalicious$(\mathsf{pk}_P, \mathsf{pk}_V, \mathsf{lpk}_P, \{h_{\mathsf{msg},i}, \mathsf{epk}_i\}_{i=1}^{M_{\mathsf{CC}}})$
2:     Define scripts:
       $\mathsf{ChallengeAssertScript} := \bigwedge_{j=1}^{2n} \Big[ \Big( \bigwedge_{i=1}^{M_{\mathsf{CC}}} \mathsf{HashLock}((\mathsf{epk}_i)_j^0) \wedge \mathsf{HashLock}(\mathsf{lpk}_j^0) \Big)$
3:         $\vee \Big( \bigwedge_{i=1}^{M_{\mathsf{CC}}} \mathsf{HashLock}((\mathsf{epk}_i)_j^1) \wedge \mathsf{HashLock}(\mathsf{lpk}_j^1) \Big) \Big]$    ▷ Replaces CheckLampSigsMatch in Alg. 3
       $\mathsf{WronglyChallengedScript} := \bigvee_{i=1}^{M_{\mathsf{CC}}} (\mathsf{HashLock}(h_{\mathsf{msg},i}))$    ▷ Replaces $\mathsf{HashLock}(h_{\mathsf{msg}})$ in Alg. 3
4:     Construct transaction skeletons $\overline{\mathsf{tx}}_{\mathsf{Deposit}}, \overline{\mathsf{tx}}_{\mathsf{Assert}}, \overline{\mathsf{tx}}_{\mathsf{ChallengeAssert}}, \overline{\mathsf{tx}}_{\mathsf{NoWithdraw}}, \overline{\mathsf{tx}}_{\mathsf{WronglyChallenged}}, \overline{\mathsf{tx}}_{\mathsf{Withdraw}}$ as in Sec. 6.1.3
5:     $\mathcal{T} := \{\overline{\mathsf{tx}}_{\mathsf{Deposit}}, \overline{\mathsf{tx}}_{\mathsf{Assert}}, \overline{\mathsf{tx}}_{\mathsf{ChallengeAssert}}, \overline{\mathsf{tx}}_{\mathsf{NoWithdraw}}, \overline{\mathsf{tx}}_{\mathsf{WronglyChallenged}}, \overline{\mathsf{tx}}_{\mathsf{Withdraw}}\}$
6:     $\mathcal{S} := \{\overline{\mathsf{tx}}_{\mathsf{Withdraw}}\}$
7:     return $(\mathcal{T}, \mathcal{S})$
8: end function
```