SoK: BitVM with Succinct On-Chain Cost

Weikeng Chen
L2 Iterative

Abstract

This is a systematization of knowledge (SoK) on BitVM with succinct on-chain cost.

1. **from different cryptographic primitives:**
   - Minicrypt privacy-free garbled circuits (PFGC)
   - homomorphic message authentication codes (HMAC), which implies succinct PFGC
   - attribute-based laconic function evaluation (AB-LFE), which implies reusable PFGC
2. **using different malicious security compilers:**
   - cut-and-choose (C&C)
   - non-interactive zero-knowledge proofs (NIZK)
   - fraud proofs on Bitcoin
3. **with different proof systems:**
   - publicly verifiable SNARK
   - designated-verifiable SNARK (DV-SNARK)
4. **in different protocol directions:**
   - standard BitVM (operator = garbler, challengers = evaluators)
   - reverse BitVM (operator = evaluator, challengers = garblers)
5. **given different operator liveness setup:**
   - existential honesty
   - honest majority

Contents

1 Introduction

BitVM [Lin23; AAL+24; LAZ+24; Lin25b] is a family of protocols for disputable computation on Bitcoin and can be used for cross-chain bridges. A challenge in BitVM2 is the on-chain cost. Babylon conducted an experiment on the Bitcoin mainnet in June 2025 [Tse25].

- **Happy path:** If the computation is correct and was not challenged, it costs $52.49.
- **Slightly unhappy path:** If the computation is correct but was challenged, it costs $4162.55.
- **Unhappy path:** If the computation is incorrect and was challenged, it costs $15,742.55.

Several factors contribute to this high cost.

- BitVM2 requires Lamport [Lam79] or Winternitz [Mer89] one-time signatures for all inputs and intermediate values of the computation, leading to a significant Bitcoin script size and gas fees. Fiamma [Fia24] showed in August 2024 that if these data can be published elsewhere for data availability, the cost can be significantly reduced.
- Non-standard transactions in BitVM2 are rejected by most Bitcoin network nodes and require the sender to work with a miner, such as MARA through Slipstream [Sli], which charges several times the gas fees for standard transactions. Moreover, the sender needs to wait until this miner mines a block, so it cannot be just an arbitrary miner.

This leads to the research of a new generation of BitVM protocols, collectively called BitVM3, that eliminates nearly all the on-chain costs, starting from Delbrag [Rub25a; Rub25b] by Jeremy Rubin, and concurrently Glock [Eag25] by Liam Eagen.

Delbrag: BitVM from GC. Jeremy Rubin showed in Delbrag that by using Yao's garbled circuits [Yao86], one can reduce the on-chain cost of BitVM to be independent of the computation, but roughly 60 bytes per input bit (using RIPEMD-160 [DBP96]). The on-chain cost becomes very manageable, and this result is likely optimal and cannot be further improved.

[PRV12] shows how to use GC for disputable computation. Consider a circuit $C : \{0,1\}^\ell \to \{0,1\}$. The computation is correct if $C(\mathbf{x}) = 1$ and incorrect if $C(\mathbf{x}) = 0$. The circuit C is garbled into Γ, and one can evaluate Γ using the input labels corresponding to $\mathbf{x}$.

The challenger receives the input labels for $\mathbf{x}$.

- If the BitVM operator is honest, $C(\mathbf{x}) = 1$ always holds, and the challenger can learn the output label for 1, which has no usage in BitVM.
- If computation is incorrect, $C(\mathbf{x}) = 0$, the challenger can learn the output label for 0. BitVM forfeits the effects of the computation when this output label is present.

If we let circuit C be a SNARK verifier and let $\mathbf{x}$ be the proof, the operator needs to know a witness that satisfies the statement that the SNARK verifier is checking to be able to find $\mathbf{x}$ such that $C(\mathbf{x}) = 1$. In BitVM use cases, the SNARK verifier usually runs a Bitcoin and Ethereum light client and checks some on-chain activities such as token transfer.

The kind of garbled circuit scheme needed for Delbrag is a privacy-free garbled circuit (PFGC) scheme [FNO15; ZRE15; KP17; BA25; LWYY25b], in which, unlike the traditional garbled circuit, the evaluator knows all the input bits. This opens up a design space with many efficient implementations. In the rest of the paper, all GCs that we refer to are by default *privacy-free*.

1.1 Tradeoffs in off-chain cost

Now that we have a succinct on-chain cost directly from GC, we need to study its off-chain cost, including computation and communication overheads.

The GC in BitVM represents an SNARK verifier. An estimate of the Boolean circuit for Groth16 verifier [Gro16] on BN-254, Citrea [Cit25], is 2.7×10^9 AND gates. Using the most efficient PFGC [ZRE15] in Minicrypt [Imp95], each GC is about 40 GB. If we use C&C and manage to require only 4 copies (see § 4.1), each GC requires 160 GB to be transferred and stored. It is possible but not ideal, especially if the GCs are broadcast to many potential challengers on the Internet. Furthermore, a protocol with BitVM needs many GCs.

- If we have multiple operators for liveness guarantees, each operator needs to have its own independent GC.
- If an operator wants to keep several BitVM instances in parallel, each instance needs to have its own GC.

One can change the off-chain cost in different ways.

Other cryptographic primitives. Minicrypt PFGC [ZRE15] has the following complexity:[1]

- communication overhead: $O_{\lambda,\kappa}(|C|)$
- garbler compute overhead: $O_{\lambda,\kappa}(|C|)$
- evaluator compute overhead: $O_{\lambda,\kappa}(|C|)$

If we instantiate PFGC with HMAC from [ILL25a; ILL25b] or [LWYY25b] (see § 3.2), then after a one-time setup, we have:

- communication overhead: $O_{\lambda,\kappa}(|\mathsf{in}|)$
- garbler compute overhead: $O_{\lambda,\kappa}(|C|)$
- evaluator compute overhead: $O_{\lambda,\kappa}(|C|)$

If we instantiate PFGC with AB-LFE with unbounded depth from [HLL23] (see § 3.3), then after a one-time setup, we have:

- communication overhead: $O_{\lambda,\kappa}(|\mathsf{in}|)$
- garbler compute overhead: $O_{\lambda,\kappa}(|\mathsf{in}|)$
- evaluator compute overhead: $O_{\lambda,\kappa}(|C|)$

But we warn that complexity does not represent concrete costs as AB-LFE and HMAC are significantly more expensive than [ZRE15] and are at most *nearly practical*.

Other malicious security compilers. C&C—batched or with DUPLO (see § 4.1)—requires each GC be replicated a number of times (related to the statistic security parameter κ), increasing communication and storage overhead. Replications can be avoided.

- C&C:
  - additional communication overhead: $O_\lambda(\kappa \cdot |C|)$

[1] $O_{\lambda,\kappa}(\cdot)$ hides a polynomial of the computational security parameter λ and statistic security parameter κ.

  - additional garbler compute overhead: $O_\lambda(\kappa \cdot |C|)$
  - additional evaluator compute overhead: $O_\lambda(\kappa \cdot |C|)$
  - additional evaluator storage overhead: $O_\lambda(\kappa \cdot |C|)$
- NIZK, assuming succinctness:
  - additional communication overhead: $O_\lambda(1)$
  - additional garbler compute overhead: $O_\lambda(|C|)$
  - additional evaluator compute overhead: $O_\lambda(|C|)$
  - additional evaluator storage overhead: $O_\lambda(1)$
- fraud proofs on Bitcoin:
  - additional communication overhead: $O_\lambda(|C|)$
  - additional garbler compute overhead: $O_\lambda(|C|)$
  - additional evaluator compute overhead: $O_\lambda(|C|)$
  - additional evaluator storage overhead: $O_\lambda(|C|)$

Although the fraud proofs on Bitcoin increase the communication and storage overhead, it is expected to be smaller than that of C&C.

Other proof systems. A Groth16 proof consists of 2 $\mathbb{G}_1$ and 1 $\mathbb{G}_2$ elements (with BN254, it equals 1016 bits). The Groth16 verifier on BN254 performs the following computation. We focus on group operations.

- $O(|\mathsf{in}|)$ scalar multiplications under fixed base, which can be removed if the public input can be hardcoded
- 2 pairings where the $\mathbb{G}_2$ can be prepared and hardcoded, one of the pairings can be removed if the public input can be hardcoded
- 1 pairing where the $\mathbb{G}_2$ cannot be prepared or hardcoded

There are many other proof systems:

- Plonk [GWC19]:
  - Proof size: 7 $\mathbb{G}_1$ and 6 $\mathbb{F}$; with BN254, it equals 3302 bits
  - 16 scalar multiplications, 8 of which are fixed-base
  - 2 pairings where the $\mathbb{G}_2$ can be prepared and hardcoded
- DV Plonk [GWC19] with [Orr25]:
  - Proof size: 7 $\mathbb{G}_1$ and 6 $\mathbb{F}$; with BN254, it equals 3302 bits
  - 17 scalar multiplications, 8 of which are fixed-base
- Fflonk [GW21]:
  - Proof size: 4 $\mathbb{G}_1$ and 15 $\mathbb{F}$; with BN254, it equals 4826 bits
  - 5 scalar multiplications, 2 of which are fixed-base
  - 2 pairings where the $\mathbb{G}_2$ can be prepared and hardcoded

- DV Fflonk [GW21] with [Orr25]:
  - Proof size: $4\ \mathbb{G}_1$ and $15\ \mathbb{F}$; with BN254, it equals 4826 bits
  - 6 scalar multiplications, 2 of which are fixed-base
- Polymath [Lip24]:
  - Proof size: $3\ \mathbb{G}_1$ and $1\ \mathbb{F}$; with BN254, it equals 1016 bits
  - 2 scalar multiplications on $\mathbb{G}_1$, 1 of which is fixed-base
  - 1 scalar multiplication on $\mathbb{G}_2$, which is fixed-base
  - 1 pairing where the $\mathbb{G}_2$ can be prepared and hardcoded
  - 1 pairing where the $\mathbb{G}_2$ cannot be prepared or hardcoded[2]
- DV Polymath [Lip24] with [Orr25]:
  - Proof size: $3\ \mathbb{G}_1$ and $1\ \mathbb{F}$; with BN254, it equals 1016 bits
  - 3 scalar multiplications on $\mathbb{G}_1$, 1 of which is fixed-base
- Pari [DMS24]:
  - Proof size: $2\ \mathbb{G}_1$ and $2\ \mathbb{F}$; with BN254, it equals 1016 bits
  - 3 scalar multiplications on $\mathbb{G}_1$, all of which are fixed-base
  - 1 scalar multiplication on $\mathbb{G}_2$, which is fixed-base
  - 2 pairings where the $\mathbb{G}_2$ can be prepared and hardcoded
  - 1 pairing where the $\mathbb{G}_2$ cannot be prepared or hardcoded[3]
- DV Pari [DMS24] with [Orr25]:
  - Proof size: $2\ \mathbb{G}_1$ and $2\ \mathbb{F}$; with BN254, it equals 1016 bits
  - 2 scalar multiplications on $\mathbb{G}_1$, 1 of which is fixed-base
- Eagen25 [DMS24; Eag25]
  - Proof size: $2\ \mathbb{G}_1$ and $1\ \mathbb{F}$; with BN254, it equals 762 bits
  - 2 scalar multiplications on $\mathbb{G}_1$, all of which are fixed-base
  - 1 scalar multiplication on $\mathbb{G}_2$, which is fixed-base
  - 1 pairing where the $\mathbb{G}_2$ can be prepared and hardcoded
  - 1 pairing where the $\mathbb{G}_2$ cannot be prepared or hardcoded[4]
- DV Eagen25 [DMS24; Eag25] with [Orr25]
  - Proof size: $2\ \mathbb{G}_1$ and $1\ \mathbb{F}$; with BN254, it equals 762 bits
  - 2 scalar multiplications on $\mathbb{G}_1$, 1 of which is fixed-base
- DV SNARK from lattices [BCI+13; ISW21] with reusability restrictions (see § 5.2):

[2]There is an alternative implementation of Polymath verifier that has 3 scalar multiplications on $\mathbb{G}_1$, two of which are fixed-base, and 3 pairings where the $\mathbb{G}_2$ can be prepared and hardcoded. This may be more efficient in practice.

[3]There is an alternative implementation of Pari verifier that has 4 scalar multiplications on $\mathbb{G}_1$, 3 of which are fixed-base, and 4 pairings where the $\mathbb{G}_2$ can be prepared and hardcoded. This may be more efficient in practice.

[4]There is an alternative implementation of Eagen25 verifier that has 3 scalar multiplications on $\mathbb{G}_1$, two of which are fixed-base, and 3 pairings where the $\mathbb{G}_2$ can be prepared and hardcoded. This may be more efficient in practice.

  - Proof size: about 11 KB for an R1CS over $\mathbb{F}_{(2^{19}-1)^2}$ with 2^{20} constraints.
  - estimated to have 5×10^5 multiplication of a 9-bit number (hardcoded and very sparse) and a 41-bit number.
- DV SNARK from Paillier [BCI+13] with [BSCG+13; Pai99; DJ01]:
  - Proof size: about 4096 bits (with packing)
  - 1 modular exponentiation in a 4096-bit field by a fixed but secret exponent of about 2048 bits with the help of Chinese Remainder Theorem (CRT) under fixed but secret primes
- DV (3-element) Groth16 with non-reusable preprocessing [Gro16] with [BHI+24]:
  - Proof size: 1 $\mathbb{G}_1$ and 3 $\mathbb{F}$; with BN254, it equals 1016 bits
  - 1 scalar multiplication on $\mathbb{G}_1$, which is fixed-base
- DV 2-element Groth16 with non-reusable preprocessing [Gro16] with [BHI+24]:
  - Proof size: 1 $\mathbb{G}_1$ and 2 $\mathbb{F}$; with BN254, it equals 762 bits
  - 1 scalar multiplications on $\mathbb{G}_1$, which is fixed-base

There are three settings here.

- **Publicly verifiable.** In this setting, proofs can be verified by anyone. The proof systems that achieve the shortest proofs, and therefore smallest on-chain cost is Eagen25:
  - Eagen25: 2 $\mathbb{G}_1$ and 1 $\mathbb{F}$.

  The proof takes 762 bits if instantiated with BN254. However, if we strive to reduce the compute overhead, we will focus on reducing the number of pairings (that can be prepared and hardcoded). One direction is Plonk and fflonk with 2 pairings, both of which can be prepared and hardcoded, but with longer proofs (3302 bits and 4826 bits):
  - Plonk: 2 pairings where the $\mathbb{G}_2$ can be prepared and hardcoded
  - Fflonk: 2 pairings where the $\mathbb{G}_2$ can be prepared and hardcoded

  However, if one can accept one of the two pairings being unable to be prepared or hardcoded, Pari and Eagen25 also work, and both have shorter proofs.
  - Polymath (1016 bits): 2 pairings, in one of which the $\mathbb{G}_2$ can be prepared and hardcoded, but not the other
  - Eagen25 (762 bits): 2 pairings, in one of which the $\mathbb{G}_2$ can be prepared and hardcoded, but not the other
- **Designated-verifier with reusable preprocessing.** In this setting, proofs can only be verified by designated verifiers who hold a secret verification key that the prover must not know. The proof systems that achieve the shortest proofs are:
  - DV Eagen25: 2 $\mathbb{G}_1$ and 1 $\mathbb{F}$

  This proof system achieves 762 bits if instantiated with BN254. It also has low off-chain cost, with 2 scalar multiplications on $\mathbb{G}_1$, 1 of which is fixed-base.
- **Designated-verifier with non-reusable preprocessing.** In this setting, the proof verification is an interactive protocol, and after verifying one proof, the preprocessing (such as SRS) cannot be reused. The proof system that achieve the shortest proofs is

- DV 2-element Groth16 with non-reusable preprocessing: $1\ \mathbb{G}_1$ and $2\ \mathbb{F}$

  It achieves 762 bits if instantiated with BN254 and only has 1 scalar multiplication on $\mathbb{G}_1$, which is fixed-base. However, note that preprocessing being non-reusable means that each time a new preprocessing needs to be done and sent out, which can take a few GBs of communication that is separate from the GC, depending on the SRS size of that DV-SNARK scheme. This is only useful if one really wants to reduce the overheads to store and to evaluate the GC.

Note that one can also use hash-based SNARK (i.e. STARK), but the proof size and verification complexity are much more complicated, and the larger proof size implies a much higher on-chain cost that may defeat the purpose of BitVM from GC. We defer the discussion to Appendix C.

Other protocol directions. The standard BitVM using GC has the operator be the garbler, and anyone can be the challengers by being the evaluators of the GC.

- **Standard BitVM:**
  - Operator generates the GC, broadcasts the GC off-chain, and publishes input labels on-chain.
  - Challengers download and verify the GC, and if the operator misbehaves, challengers evaluate the GC with input labels and publish the output label on-chain.

The burden is that the challengers must store the GC and evaluate the GC as soon as possible, within the challenge period, when the operator misbehaves. For user experience and liquidity turnover, we want to minimize this challenger period as much as possible. If we assume designated verifiers, we can have an alternative construction called reverse BitVM, which shifts most of the burden from the challengers to the operator. Reverse BitVM was implicit in Delbrag [Rub25b].

- **Reverse BitVM:**
  - Each of the challengers generates the GC and sends the GC to the operator off-chain. If the operator misbehaves, at least one of the challengers forces the operator to start a label exchange protocol (see § 2.4) with the operator on-chain.
  - Operator downloads and verifies the GC, and upon being challenged by a challenger, participates in the label exchange protocol with this challenger, evaluates the GC with input labels, and publishes the output label on-chain.

The benefits of reverse BitVM are as follows:

- Challengers do not need to store the GC after being verified by the operator, but only the input labels, which are small.
- Challengers do not need to evaluate the GC.

The downsides of reverse BitVM are as follows:

- The operator needs to do more work, which grows linearly to the number of challengers.
- BitVM becomes inherently designated-verifier.
- The overall protocol and transaction workflow becomes much more complicated.

Author's Note

New readers may find it confusing to understand standard BitVM and reverse BitVM altogether, especially because reverse BitVM is more intricate. It is recommended to skip reverse BitVM entirely during the first read.

Another limitation is that reverse BitVM (§ 2) with designated-verifier SNARK with reusable preprocessing (§ 5) is not compatible with privacy-free GC (PFGC) but requires standard GC.[5] In [ZRE15], it at most doubles the computational and communication overhead of the garbler and the evaluator compared to PFGC. However, this restriction excludes AB-LFE and HMAC, as they can only do PFGC but not standard GC. Succinct garbling [LWYY25b; ILL25b; ILL25a] and reusable garbling [GKP+13] exist, but at much more significant computation overheads.

Other operator liveness setup. The standard liveness setup from the BitVM bridge [LAZ+24] is to assume existential honesty (while safety is guaranteed by the challengers): as long as one of the operators participates in the protocol, the asset locked in BitVM can be released.

In both standard BitVM and reverse BitVM, different operators have separate GCs. When there are N operators:

- **Standard BitVM:**
  - Each operator generates its own GC.
  - Challengers download and verify GCs from all operators, and need to prepare to challenge all operators in the worst case, which requires evaluating all the GCs.
- **Reverse BitVM:**
  - Each challenger needs to generate GC for each operator.
  - Each operator downloads and verifies the GCs from each challenger and needs to prepare to respond to all challengers at the same time in the worst case, which requires evaluating all the GCs of the challengers.

There is a different setup which requires a majority of operators to be online for liveness, while safety is still guaranteed by the challengers.

For standard BitVM, we do not find advantages of this new setup over the setup with existential honesty because it seems to still require N copies of GCs. There are ways for multiple operators to collectively generate *one* GC such that only a majority of operators can learn the input labels:

- secure multiparty computation (MPC) [GMW87; BGW88]
- rerandomizable garbling [GHV10; HHJ25] (see § 7.2)
- learning parity with noise (LPN)-based multiparty garbling [BCO+21; BGH+23]

But, when there are fewer than 100 operators, the computation and communication overheads are *not* competitive with having each operator own a separate GC, which would be the same outcome as in the previous setup with existential honesty. However, honest majority is useful in reverse BitVM as it can reduce the number of GCs to be generated and communicated.

[5] The standard GC can also be reaplced with a mix (called partial garbling) of a PFGC and a standard GC, where the PFGC depends on data that the evaluator (here the operator) can know.

- **Reverse BitVM:**
  - Each challenger generates *one* GC for all the operators collectively.
  - All operators download and verify the GCs from each challenger. They need to prepare to respond to all challengers at the same time in the worst case, which requires requesting the input labels collectively and evaluating all the GCs of the challengers.
  - All operators secret-share a signing key or use Bitcoin script multisig to enforce that only a honest majority can respond to the challengers.

The benefit is that, for each challenger:

- Previously, each challenger needs to generate one GC for each operator, now, it only needs to generate one GC for all the operators collectively.
- Previously, each challenger needs to send a different GC to each operator. Now, assuming that the operators are more powerful and can help, the challenger can send the GC to one operator and have this operator share the GC with the rest of the operators. To prevent this operator from manipulating the GC, the challenger can share the GC's hash digest with the other operators.

However, each operator still downloads and stores one GC from each challenger, although only one of the (honest) operators needs to evaluate the GC.

The key downside of this alternative liveness setup is that the protocol now requires an honest majority of operators for liveness.

1.2 Handling the off-chain cost

If one has made their selection after studying the tradeoffs, but the off-chain cost is still significant, there are some ways to manage and handle the off-chain cost.

Use BitVM less frequently. The most significant use case for BitVM is to build a native bridge from Bitcoin to another chain—without loss of generality, Ethereum as the example—and issue wrapped BTC. BitVM provides two important guarantees.

- **Peg-in:** Wrapped BTCs minted on Ethereum in this way are pegged in by locking the same amount of BTCs on Bitcoin. This sets the upper bound of the price of wrapped BTCs.
- **Peg-out:** Anyone with wrapped BTCs on Ethereum can withdraw them to unlock the BTCs previously locked on Bitcoin. This sets the lower bound of the price of wrapped BTCs.

A trust-minimized peg-out procedure is important because peg-in alone does not prevent depegging, and many alternatives to BitVM do not offer peg-out in the trust-minimized way.

To enjoy the benefits of BitVM in such a cross-chain bridge for BTC (i.e. wrapped BTC), however, one does not need to use BitVM all the time.

Protocol 1: TierNolan atomic swap [Tie13]

Below is a rephrased version of the TierNolan atomic swap protocol, using a new opcode (`OP_CHECKSEQUENCEVERIFY`) that became available after TierNolan's 2013 post.

- Alice has w_1 BTCs on Bitcoin
- Bob has w_2 wrapped BTCs on Ethereum

They can perform an atomic swap, with the following protocol.

- **Alice:**
  - Sample a random string $s \in \{0,1\}^\lambda$ where λ is the security parameter. For simplicity, we can assume $\lambda = 128$. Compute its SHA-256 hash $h = H(s)$.
  - Create a Bitcoin script as follows:

```
OP_SHA256 <h> OP_EQUAL
OP_IF
      <Bob pubkey>
OP_ELSE
      <t_1> OP_CHECKSEQUENCEVERIFY OP_DROP
      <Alice pubkey>
OP_ENDIF
OP_CHECKSIG
```

    where t_1 is the wait time until cancellation, which can be a few hours.
  - Deposit w_1 BTC into the Bitcoin script above and send h to Bob.
- **Bob:**
  - Verify Alice's deposit. Create a smart contract on Ethereum that has the same functionality as the Bitcoin script above with the same h but potentially a different wait time t_2, and with the two public keys swapped.[a]
  - Deposit w_2 wrapped BTC into this smart contract.

Now they can withdraw the tokens:

- Alice invokes that smart contract on Ethereum, to withdraw w_2 wrapped BTC showing the knowledge of s such that $h = H(s)$.
- After Alice shows s, Bob withdraws w_1 BTC using s on Bitcoin.

If the protocol was halted in the middle,

- On Bitcoin, Alice waits t_1 and invokes the Bitcoin script with any dummy hash to refund the w_1 BTCs.
- On Ethereum, Bob waits t_2 and invokes the smart contract with any dummy hash to refund the w_2 wrapped BTCs.

[a] In practice, Bob does not need to create a new smart contract, but can reuse an existing public, verified smart contract that simulates hashed timelock contracts (HTLC), e.g. [Hat].

- BitVM should only be used to add or remove liquidity from wrapped BTC *in large quantities*. This is usually done by liquidity providers or market makers, not regular users.
- Users who want to bridge wrapped BTCs can use the TierNolan atomic swap [Tie13] (shown in Protocol 1), introduced back in 2013, to work with a market maker on the other side to perform a fully trustless atomic swap, without involving BitVM.

In this way, BitVM is used infrequently and is used usually in a planned and managed fashion with a handful of parties that can work together. For example, liquidity providers and market makers can join as one of the operators if they are concerned about liveness and can join as one of the challengers if they are concerned about safety. All of this can be done in a way that removes the counter-party risk.

Try to keep the GC reusable. A GC can be reused again in BitVM if it satisfies two conditions:

- None of the input labels corresponding to the proof have been revealed to the evaluators (in standard BitVM, the challengers; in reverse BitVM, the operator)
- The GC is currently not required for any ongoing BitVM instances.

In the BitVM protocol (§ 2), the input labels are revealed:

- **Standard BitVM:**
  - When a challenger sends out the Challenge transaction, the operator is required to respond with the Assert transaction, which reveals the input labels.
- **Reverse BitVM:**
  - When a challenger sends out the Challenge transaction, the operator is required to respond with the SubmitSolution transaction. The challenger might (or might not) further responds with the InputLabels transaction, which reveals the input labels (see § 2.3).

We can try to avoid revealing the input labels as follows.

- **Operator should try to convince the challengers off-chain**. The operator must facilitate enough information off-chain, in a timely manner to the challengers so that they can verify the computation without the need to even start the Challenge transaction. In reverse BitVM, challengers can terminate the challenge protocol early if convinced by the operator off-chain or from the proof in the SubmitSolution transaction, by not responding with the InputLabels transaction, to avoid revealing the input labels of the GC.
- **Challengers should pay a collateral to challenge.** An idea from the BitVM bridge paper [LAZ+24] is to require the challenger to pay a collateral when submitting the Challenge transaction (or the InputLabels transaction), which covers the cost of the GC in case the challenger is malicious. Previously, this collateral may need to be at least $4,000 [Tse25] due to the high gas cost of the Assert transaction in [LAZ+24], but now, the cost is only about the generation of GC, which is significantly lower and more reasonable for the challengers.

Transfer and store GC in the right way. BitVM from Minicrypt PFGC has a very practical compute overhead, but it also has significant network and storage overheads. We now discuss how to address these overheads in the right way.

- **Network:** If the operator and challengers are located in clouds or data centers, one should try to utilize the available bandwidth (see Skyplane [JKW+23]). Otherwise, remember Andrew Tanenbaum's quote *"Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway."*. One can either use Azure Data Box, Google Transfer Appliance, or AWS Snowball to transmit large amounts of data or simply ship the SSDs in express mail.
- **Storage:** Cloud storage providers offer the coldest storage (also known as glacier storage) at a low price, which is suitable for GCs since the accesses are expected to be rare. If GCs are stored locally, the price of SSDs has gone down significantly in the past decade.

2 BitVM

In this section, we discuss how GC changes the protocol in the BitVM2 bridge paper [LAZ+24] to have a succinct on-chain cost. We start by providing the necessary background of GC.

2.1 Background: GC

The verification algorithm of a proof system can be represented as a Boolean circuit $C(\mathbf{x}) : \{0,1\}^\ell \to \{0,1\}$ that takes ℓ bits of input (which is the proof and some other necessary information) and outputs a bit indicating whether the proof is valid or not.

A privacy-free garbling scheme (which includes all instantiations, from Minicrypt PFGC, AB-LFE, or HMAC) for circuit C has the following syntax:

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{gk})$: The setup algorithm generates the public parameters pp and the garbling key gk for the garbler.
- $\mathsf{Garble}(\mathsf{gk}) \to (\Gamma, \{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}, L_o^{(0)}, L_o^{(1)})$: The garbling algorithm outputs the GC Γ, both input labels for each input bit $\{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}$, and both labels for the output $L_o^{(0)}$ and $L_o^{(1)}$.
- $\mathsf{Evaluate}(\mathsf{pp}, \Gamma, \mathbf{x}, \{L_i^{(x_i)}\}_{i\in[\ell]}) \to L_o^{(b_o)}$: The evaluation algorithm takes the public parameters pp, the GC Γ, the input $\mathbf{x}$, and one of the input labels for each input bit $\{L_i^{(x_i)}\}_{i\in[\ell]}$. If the labels are correct, it outputs $L_o^{(0)}$ if $C(\mathbf{x}) = 0$ and $L_o^{(1)}$ if $C(\mathbf{x}) = 1$.

Privacy-free GC schemes are called "privacy-free" because the Evaluate algorithm requires the knowledge of the input $\mathbf{x}$. Standard GC with privacy guarantees does not require so and can work with an evaluator who only knows the input labels of $\mathbf{x}$, but not the concrete value of $\mathbf{x}$ itself.

Correctness. We say that the GC scheme is correct if the evaluation algorithm correctly evaluates the circuit.

$$\mathsf{Evaluate}(\mathsf{pp}, \Gamma, \mathbf{x}, \{L_i^{(x_i)}\}_{i\in[\ell]}) = L_o^{(C(\mathbf{x}))}$$

Authenticity. We say that the GC scheme provides authenticity if, for an efficient adversary $\mathcal{A}$, given $\mathsf{pp}, \Gamma, \mathbf{x}, \{L_i^{(x_i)}\}_{i\in[\ell]}$ such that $C(\mathbf{x}) = b_o$, it follows that $\mathcal{A}$ can learn $L_o^{(b_o)}$, but $\mathcal{A}$ cannot learn $L_o^{(1-b_o)}$. Formally, consider $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, it is defined as follows:

$$\mathbf{Pr}\left[\mathcal{A}_2(\mathsf{st}, \{L_i^{(x_i)}\}_{i\in[\ell]}) = L_o^{(1-C(\mathbf{x}))} \;\middle|\; \begin{array}{l}(\mathsf{pp}, \mathsf{gk}) \leftarrow \mathsf{Setup}(1^\lambda)\\ (\Gamma, \{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}, L_o^{(0)}, L_o^{(1)}) \leftarrow \mathsf{Garble}(\mathsf{gk})\\ (\mathsf{st}, \mathbf{x}) \leftarrow \mathcal{A}_1(\Gamma)\end{array}\right] = \mathsf{negl}(\lambda)$$

where $\mathsf{negl}(\lambda)$ is a negligible probability function. This means that:

- **Standard BitVM:**
  - If a challenger has the input labels for a correct proof (such that $C(\mathbf{x}) = 1$), a challenger can get $L_o^{(1)}$, but the challenger can never get $L_o^{(0)}$ (which is used for challenging).
- **Reverse BitVM:**
  - If the operator has the input labels for an incorrect proof (such that $C(\mathbf{x}) = 0$), the operator can get $L_o^{(0)}$, but the operator can never get $L_o^{(1)}$ (which is used to reject the challenge).

2.2 Prepare the GC

Before Alice deposits money in the BitVM protocol, M operators and N challengers need to prepare the GC, assuming that the setup algorithm for GC has already been completed. The roles of operators and challengers differ in standard BitVM and reverse BitVM.

If we assume existential honesty for the operator liveness setup:

- **Standard BitVM:**
  - Each of the M operators is a garbler of its own GC.
    - The wall clock time for each operator is $O_{\lambda,\kappa}(|C|)$.
    - The total compute overhead is $O_{\lambda,\kappa}(M \cdot |C|)$.
    - The storage overhead for each operator is $O_{\lambda,\kappa}(|\mathsf{in}|)$, in total $O_{\lambda,\kappa}(M \cdot |\mathsf{in}|)$.
  - Each of the N challengers downloads and verifies the M GCs. Each of them is an evaluator for all M GCs. Everyone in the public can be the challengers unless explicitly made permissioned.
    - The wall clock time for each challenger is $O_{\lambda,\kappa}(M \cdot |C|)$.
    - The total computation and communication overheads are $O_{\lambda,\kappa}(MN \cdot |C|)$.
    - The storage overhead for each challenger is $O_{\lambda,\kappa}(M \cdot |C|)$, in total $O_{\lambda,\kappa}(MN \cdot |C|)$.
- **Reverse BitVM:**
  - Each of the N challengers is a garbler for M GCs, one for each operator.
    - The wall clock time for each challenger is $O_{\lambda,\kappa}(M \cdot |C|)$.
    - The total compute overhead is $O_{\lambda,\kappa}(MN \cdot |C|)$.
    - The storage overhead for each challenger is $O_{\lambda,\kappa}(M \cdot |\mathsf{in}|)$, in total $O_{\lambda,\kappa}(MN \cdot |\mathsf{in}|)$.
  - Each of the M operators downloads and verifies the N GCs for it.
    - The wall clock time for each operator is $O_{\lambda,\kappa}(N \cdot |C|)$.
    - The total computation and communication overheads are $O_{\lambda,\kappa}(MN \cdot |C|)$.
    - The storage overhead for each operator is $O_{\lambda,\kappa}(N \cdot |C|)$, in total $O_{\lambda,\kappa}(MN \cdot |C|)$.

If we assume an honest majority for operators, which is only interesting in reverse BitVM:

- **Reverse BitVM:**
  - Each of the N challengers is a garbler for one GC, collectively for all operators.
    - The wall clock time for each challenger is $O_{\lambda,\kappa}(|C|)$.
    - The total compute overhead is $O_{\lambda,\kappa}(N \cdot |C|)$.
    - The storage overhead for each challenger is $O_{\lambda,\kappa}(|\mathsf{in}|)$, in total $O_{\lambda,\kappa}(N \cdot |\mathsf{in}|)$.
  - Each of the M operators downloads and verifies the N GCs for it.
    - The wall clock time for each operator is $O_{\lambda,\kappa}(N \cdot |C|)$.
    - The total computation and communication overheads are $O_{\lambda,\kappa}(MN \cdot |C|)$.
    - The storage overhead for each operator is $O_{\lambda,\kappa}(N \cdot |C|)$, in total $O_{\lambda,\kappa}(MN \cdot |C|)$.

We require each garbler to additionally send the following associated hashes:[6]

- A SHA-256 or RIPEMD-160 hash of each of the labels (one for 0, one for 1) for each input bit.
- A SHA-256 or RIPEMD-160 hash of each of the labels (one for 0, one for 1) for the output bit.

Input labels longer than 128 bits can be compressed using a technique in the BitVM3 paper [Lin25b], in which the garbler encrypts the label under a per-label short key, proves the correctness of the encryption, and uses the short keys instead of the labels in the rest of the protocol.

The correctness of the GC and the associated hashes are later handled by C&C, NIZK, and/or fraud proofs on Bitcoin. See § 4 for more discussion.

2.3 On-chain protocol

We start with standard BitVM with the classic 8-transaction BitVM transaction flow in the BitVM bridge paper [LAZ+24]. We ignore the transaction fees in the following protocol descriptions, but a practical implementation should adjust the numbers based on a reasonable estimate of the transaction fees. Sometimes, the Bitcoin mainnet can reach a fee rate below 1 sat per vByte.

Standard BitVM:

- **PegIn:**
  - This transaction takes v ₿ from Alice and locks it, until it is unlocked later during withdrawal by some **PayoutOptimistic-i** transaction or some **Payout-i** transaction.
  - This transaction is presigned by the signing committee to emulate a covenant.
- **PegOut-i:**
  - This transaction takes $(v - f_O)$ ₿ from the i-th operator where f_O is a fixed fee charged by the operator and sends $(v - f_O)$ ₿ to Bob.
  - Note that [LAZ+24] has a way to prevent the race condition in which more than one operator ends up sending $(v - f_O)$ ₿ to Bob, using an existing 0 ₿ UTXO from Bob (acknowledged by smart contracts on Ethereum to be the peg-out point) with a template signature from Bob under `SIGHASH_SINGLE | ANYONECANPAY`. This transaction is NOT presigned by the signing committee.
- **Claim-i:**
  - This transaction takes d ₿ from the i-th operator and makes two UTXOs:
    - one UTXO with d ₿ that will be spent by an **Assert-i** or a **PayoutOptimistic-i** transaction
    - one UTXO with 0 ₿ that will be spent by an **Challenge-i, j** or a **PayoutOptimistic-i** transaction, a template signature on which should be made available to allow anyone to spend it under some conditions through the **Challenge-i, j:** transaction
  - This transaction is presigned by the signing committee to emulate a covenant and requires a signature from the i-th operator.

[6] We assume that the GC algorithm does not use SHA-256 or RIPEMD-160 to avoid potential composability issues and complications in the security proofs.

- **Challenge-i, j:**
  - This transaction takes the 0 ₿ UTXO from the **Claim-i** transaction and c ₿ from the j-th challenger and sends c ₿ to the i-th operator, where c is the collateral for challenging, and in the BitVM bridge paper [LAZ+24], this can be crowdfunded.
  - This transaction is NOT presigned by the signing committee, but is presigned by the i-th operator with a template signature under `SIGHASH_SINGLE | ANYONECANPAY`.
- **Assert-i:**
  - This transaction takes the d ₿ UTXO from **Claim-i**, reveals the input labels that correspond to a ZK proof, and outputs a UTXO with d ₿ that would be spent by a **Payout-i** or a **Disprove-i, j** transaction.
  - The Bitcoin script will check that the revealed input labels match the hashes computed in the preparation phase. A template signature needs to be made available for the case when this UTXO is used in the **Disprove-i, j** transaction.
  - This transaction is presigned by the signing committee and requires a signature from the i-th operator.
- **PayoutOptimistic-i:**
  - This transaction takes:
    - the v ₿ UTXO from **PegIn**
    - the d ₿ UTXO from **Claim-i**, after a relative time Δ_B has passed (Δ_B is the waiting period for any challenger to send the **Challenge-i, j** transaction)
    - the 0 ₿ UTXO from **Claim-i**

    and outputs $(v + d)$ ₿ to the i-th operator.
  - This transaction is presigned by the signing committee and requires a signature from the i-th operator.
- **Payout-i:**
  - This transaction takes:
    - the v ₿ UTXO from **PegIn**
    - the d ₿ UTXO from **Assert-i:**, after a relative time Δ_A has passed (Δ_A is the waiting period for any challenger to send the **Disprove-i, j** transaction)

    and outputs $(v + d)$ ₿ to the i-th operator.
  - This transaction is presigned by the signing committee and requires a signature from the i-th operator.
- **Disprove-i, j:**
  - This transaction takes the d ₿ UTXO from **Assert-i** and outputs b ₿ to void ("burn") and a ₿ to the j-th challenger as reward. It requires the j-th challenger to demonstrate that the proof shown in **Assert-i** is invalid.
  - This transaction is NOT presigned by the signing committee, but it uses a template signature from the signing committee that restricts that the b ₿ must be burnt from the d ₿ using `SIGHASH_SINGLE`.

Reverse BitVM is more difficult as there are multiple challengers, and each challenger uses a different GC for each operator. There are a few differences.

Reverse BitVM:

- **PegIn:**
  - This transaction takes v ₿ from Alice and locks it, until it is unlocked later during withdrawal by some **Payout-i** transaction.
  - This transaction is presigned by the signing committee to emulate a covenant.
- **PegOut-i:**
  - This transaction takes $(v - f_O)$ ₿ from the i-th operator where f_O is a fixed fee charged by the operator and sends $(v - f_O)$ ₿ to Bob.
  - It uses the same technique as in standard BitVM to avoid the race condition.
  - This transaction is NOT presigned by the signing committee.
- **Claim-i:**
  - This transaction takes d ₿ from the i-th operator and makes $(N + 1)$ UTXOs:
    - one UTXO with d ₿ that will be spent by one of these transactions.
      - one **NoSolution-i, j** transaction for some $j \in [N]$
      - one **NotProven-i, j** transaction for some $j \in [N]$
      - a **Payout-i** transaction
    - one UTXO with 0 ₿ for ***each*** of the N challengers. The j-th UTXO will be spent by one of these transactions:
      - a **NoSolution-i, j** transaction
      - a **NotProven-i, j** transaction
      - a **Payout-i** transaction
  - This transaction is presigned by the signing committee and requires a signature from the i-th operator.
- **Challenge-i, j:**
  - This transaction takes c ₿ from the j-th challenger and creates a UTXO with c ₿, where c is the collateral for challenging. Unlike standard BitVM and the BitVM bridge [LAZ+24], this collateral must be fixed before the signing committee signs the transaction and not move.[7]
    - If the challenger moves this collateral, the challenger loses the ability to challenge in this BitVM instance.
    - It is okay for the challenger to move the collateral if the challenger has been convinced by the operator off-chain that the claim is correct of if the BitVM instance has already been challenged (by some **NoSolution-i, j** or **NotProven-i, j** transaction).

    The output, which is a UTXO with c ₿, can be spent by one of the two transactions:

[7] There are ways to allow the collateral to be locked only when the j-th challenger actually wants to challenge, but we feel that the design is much more complicated.

    - a **SubmitSolution-i, j** transaction
    - a **NoSolution-i, j** transaction
  - This transaction is presigned by the signing committee and requires a signature from the j-th challenger.
- **SubmitSolution-i, j:**
  - This transaction takes the c ₿ UTXO from the **Challenge-i, j** transaction and creates a UTXO with c ₿ that can be spent by one of the two transactions:
    - a **NoResponse-i, j** transaction
    - a **InputLabels-i, j** transaction
  - This transaction requires the i-th operator to submit the *reverse input labels* (discussed in § 2.4) that correspond to the input bits. This transaction is presigned by the signing committee.
- **NoSolution-i, j:**
  - This transaction takes:
    - the d ₿ UTXO from the **Claim-i**
    - the j-th 0 ₿ UTXO from the **Claim-i**
    - the c ₿ UTXO from the **Challenge-i, j** transaction, after a relative time Δ_D has passed (Δ_D is the waiting period for the i-th operator to send the **SubmitSolution-i, j** transaction)

    and creates the following UTXOs:
    - b ₿ UTXO to void ("burn")
    - $(d + c - b)$ ₿ UTXO to the j-th challenger
  - This transaction is presigned by the signing committee.
- **InputLabels-i, j:**
  - This transaction takes the c ₿ UTXO from the **SubmitSolution-i, j** transaction and creates a UTXO with c ₿ that can be spent by one of the two transactions:
    - a **NotProven-i, j** transaction
    - a **Prove-i, j** transaction
  - This transaction is presigned by the signing committee.
- **NoResponse-i, j:**
  - This transaction takes the c ₿ from the **SubmitSolution-i, j** transaction, after a relative time Δ_C has passed (Δ_C is the waiting period for the j-th challenger to send the **InputLabels-i, j** transaction) and sends c ₿ to the i-th operator.
  - This transaction is NOT presigned by the signing committee.
- **Prove-i, j:**
  - This transaction takes the c ₿ from the **InputLabels-i, j** transaction and sends c ₿ to the i-th operator. This transaction requires the i-th operator to show the desired output label of the GC that means that the proof is valid.

  - This transaction is presigned by the signing committee and optionally requires a signature from the i-th operator.
- **NotProven-i, j:**
  - This transaction takes:
    * the d ₿ UTXO from the **Claim-i**
    * the j-th 0 ₿ UTXO from the **Claim-i**
    * the c ₿ from the **InputLabels-i, j** transaction, after a relative time Δ_B (Δ_B is the waiting period for the i-th operator to send the **Prove-i, j** transaction)

    and sends c ₿ to the j-th challenger.
  - This transaction is presigned by the signing committee.
- **Payout-i:**
  - This transaction spends $(N+2)$ UTXOs:
    * the v ₿ UTXO from **PegIn**
    * the d ₿ UTXO from the **Claim-i**, after a relative time Δ_A (Δ_A is the waiting period for all the challengers to finish all the necessary challenging interactions with the operator)
    * all the 0 ₿ UTXOs from the **Claim-i**

    and outputs $(v+d)$ ₿ to the i-th operator.
  - This transaction is presigned by the signing committee.

We will now discuss the overall workflow and the interplay between GC and the on-chain protocol.

2.4 Workflow

We start with the workflow in standard BitVM. Given the GC, the signing committee (which may consist of the operators and some challengers) generates the transactions and presigns them. The Bitcoin scripts in some of these transactions require certain operations involving GC.

Standard BitVM:

- **Assert-i:** Given the hashes for each input labels for each input bit, denoted by $\{h_k^{(0)}, h_k^{(1)}\}_{k\in[\ell]}$, the Bitcoin script enforces that the i-th operator must reveal either $L_k^{(0)}$ or $L_k^{(1)}$ for each input bit ($k \in [\ell]$), which is checked in the Bitcoin script against the hashes.
- **Disprove-i, j:** Given the hashes of each output labels, denoted by $\{h_o^{(0)}, h_o^{(1)}\}$, the Bitcoin script enforces that the j-th challenger must reveal $L_o^{(0)}$, which corresponds to $h_o^{(0)}$, that shows that the proof is invalid.

In the reverse BitVM, the i-th operator will generate *reverse input labels* $\{R_k^{(0)}, R_k^{(1)}\}_{k\in[\ell]}$ and reveal their hashes $\{r_k^{(0)}, r_k^{(1)}\}_{k\in[\ell]}$. The reverse input labels are generated independently of the actual input labels and are only used to exchange input labels.

Reverse BitVM:

- **SubmitSolution-i, j:** Given the hashes for each *reverse* input labels for each input bit, denoted by $\{r_k^{(0)}, r_k^{(1)}\}_{k\in[\ell]}$, the Bitcoin script enforces that the i-th operator must reveal either $R_k^{(0)}$ or $R_k^{(1)}$ for each input bit ($k \in [\ell]$), which is checked in the Bitcoin script against the hashes.
- **InputLabels-i, j:** Given the hashes for each input label and each *reverse* input labels for each input bit, aka $\{h_k^{(0)}, h_k^{(1)}\}_{k\in[\ell]}$ and $\{r_k^{(0)}, r_k^{(1)}\}_{k\in[\ell]}$, the Bitcoin script enforces that the j-th challenger must reveal either $(R_k^{(0)}, L_k^{(0)})$ or $(R_k^{(1)}, L_k^{(1)})$ for each input bit ($k \in [\ell]$), which is checked in the Bitcoin script against the hashes.
- **Prove-i, j:** Given the hashes of each output labels, denoted by $\{h_o^{(0)}, h_o^{(1)}\}$, the Bitcoin script enforces that the i-th operator must reveal $L_o^{(1)}$, which corresponds to $h_o^{(1)}$, that shows that the proof is valid.

When GC is ready and the transactions are presigned, user Alice can deposit v ₿ into the BitVM protocol by the **PegIn** transaction. The BitVM instance stays this way until the protocol decides to release the v ₿ to user Bob (Bob may or may not be Alice).

To release the v ₿, an operator sends the **PegOut-i** transaction to Bob. This can be done almost instantly in the next Bitcoin block, without any waiting period.

Assume that it is the i-th operator who succeeds to send the **PegOut-i** transaction. Since **PegOut-i** is designed to be exclusive, there can only be one operator that pays Bob with **PegOut-i**. However, a malicious operator who did not pay Bob may want to claim v ₿, which BitVM is designed to prevent.

The rest of the BitVM protocol is for this operator to claim the v ₿ locked in the BitVM protocol. It starts by sending the **Claim-i** transaction, both for standard BitVM and reverse BitVM.

At the same time, the i-th operator should share information off-chain to potential challengers to convince them that the i-th operator has correctly performed the **PegOut-i** transaction. Sometimes, this can be self-explanatory and obvious, as the **PegOut-i** transaction can be simply found on the Bitcoin network.

If a challenger feels that the i-th operator did not behave correctly, the challenger can initiate the challenge protocol by sending the **Challenge-i, j** transaction, both for standard BitVM and reverse BitVM. Then, the protocol continues as follows.

Standard BitVM:

- Wait until the i-th operator sends the **Assert-i** transaction or until another operator (say i'-th) has finalized the payout for this BitVM instance, by sending a **PayoutOptimistic-i'** or a **Payout-i'** transaction.
- If the i-th operator sends the **Assert-i** transaction, the j'-th challenger (j' may or may not be j) obtains the input labels and runs the GC, which is expected to have $L_o^{(0)}$ that shows that the proof is invalid. The j'-th challenger sends the **Disprove-i, j'** transaction.

Reverse BitVM:

- The j-th challenger waits at most Δ_D for the i-th operator to send the **SubmitSolution-i, j** transaction. If no such transaction appears after a period of Δ_D has passed, the j-th challenger

sends the NoSolution-i, j transaction to reject the i-th operator's claim. Otherwise, continue the workflow.

- The j-th challenger examines the reverse input labels in the SubmitSolution-i, j transaction and generates the InputLabels-i, j transaction accordingly. If after Δ_C, the i-th operator still has not seen the InputLabels-i, j transaction, the i-th operator sends the NoResponse-i, j transaction to reject the j-th challenger's challenge. Otherwise, continue the workflow.
- The i-th operator can run the GC with the input labels to obtain $L_o^{(1)}$ that shows that the proof is valid. The j-th challenger waits at most Δ_B to see the Prove-i, j transaction sent by the i-th operator, which cancels the j-th challenger's challenge. If no such transaction appears after a period of Δ_B has passed, the j-th challenger can send the NotProven-i, j transaction to reject the i-th operator's claim. Otherwise, continue the workflow.

If the i-th operator was not challenged by anyone or if all challenges have been rejected, then the operator can claim the v ₿ from PegIn and the d ₿ as deposit from Claim-i.

- **Standard BitVM:**
  - If the Assert-i transaction was never sent, the i-th operator can claim v ₿ by sending the PayoutOptimistic-i transaction.
  - If the Assert-i transaction was sent but no Disprove-i, j' was sent, the i-th operator can claim v ₿ by sending the Payout-i transaction.
- **Reverse BitVM:**
  - After Δ_A has passed, regardless of any ongoing challenges, the i-th operator can claim v ₿ by sending the Payout-i transaction.

This concludes the BitVM protocol.[8]

[8] A remaining issue in standard BitVM is to reimburse the j-th challenger who sent the Challenge-i, j transaction, and this might not be the same challenger who sent the Disprove-i, j' transaction. It requires a separate protocol or permissioned challengers (where challengers use transactions presigned by the signing committee, and the fund is released to the challengers if the i-th operator's claim gets rejected or if another operator supersedes).

3 Cryptographic primitives

In this section, we talk about how to instantiate PFGC from different cryptographic primitives: PFGC from Minicrypt [Imp95], AB-LFE [QWW18; HLL23], and HMAC [ILL25b; ILL25a].

3.1 From Minicrypt PFGC

We start with PFGC from Minicrypt, which means that they only use symmetric key primitives.

- The GC usually grows with the size of the Boolean circuit $|C|$. Specifically, the best known construction by Zahur, Rosulek, and Evans [ZRE15] requires 128 bits for each AND gate, while NOT and OR gates are free.[9]
- The GC can be generated and evaluated very efficiently. Specifically, the best known construction by Zahur, Rosulek, and Evans [ZRE15] requires 2 calls to the hash function by the garbler and 1 call by the evaluator for each AND gate, while NOT and OR gates are free.

Infeasibility of "size-zero" PFGC. An important work for Minicrypt PFGC is from Kondi and Patra [KP17], which showed that PFGC for *Boolean formula* can be made even cheaper ("size-zero") in that an AND gate requires no communication or hash function invocations. However, the Boolean formula is a very restricted representation that only allows Boolean circuits with fan-out-1 gates, and each input wire can only be used twice. It is unlikely that we can directly use such a "size-zero" PFGC for a ZK proof verifier. AuthOr [BA25] by Biçer and Ajorian showed that one can combine [KP17] with [ZRE15], but the improvement over [ZRE15] could be limited.

Optimization. A tool that can be useful for Minicrypt PFGC is garbled one-hot encoding [HK21] based on puncturable PRF [GGM84], which can improve performance on some basic operations such as lookup tables, which can be useful for fixed-base scalar multiplication,.

Advantage with free-XOR. Modern Minicrypt PFGC uses the free-XOR technique [KS08] such that XOR gates do not contribute to the size of the GC and each XOR gate can be evaluated with only XOR over the labels, which is very efficient. This is especially useful for operations over binary curves [Kob91], as used in Glock [Eag25], because field multiplication results in a lot of XOR gates but only very few AND gates. If the XOR gates are free, then binary curves are significantly cheaper than common prime-order curves.

Augmentation with privacy. As we discussed in § 1.1, in the case of DV-SNARK with reusable preprocessing in the reverse BitVM model, the evaluator (who is the operator) must not learn the verifier's secret of DV-SNARK. Since the verifier's secret is given as input to the GC, it requires a GC scheme that takes input labels for the verifier's secret, but does not require the evaluator to know that secret. PFGC does not satisfy this requirement, and instead, we would need standard GC as in [ZRE15; RR21]. This increases the cost of the GC to about $2\times$. It is possible to use partial garbling [ILL25b] so that part of the GC is still garbled using PFGC to reduce the total circuit size.

[9] Note that the hash function H in [ZRE15] needs to be changed to different ones [GKWY20; GKW+20] for security.

3.2 From HMAC

HMAC with suitable properties implies succinct partial garbling [ILL25b], which immediately implies succinct PFGC, meaning that the communication overhead, mostly $|\Gamma|$, does not grow linearly with the size of the circuit $|C|$.

Compared with Minicrypt PFGC, the communication overhead is much lower, but the compute overhead would be significantly higher because each gate involves somewhat homomorphic encryption, while [ZRE15] or [FNO15] only require hash functions.

We present the detailed constructions in Appendix A and we keep only the high-level idea here.

HMAC. Consider the following syntax for HMAC for circuit $C : \{0,1\}^\ell \to \{0,1\}$.

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{sk}, \mathsf{evk})$: the key generation algorithm takes the security parameters 1^λ as input and outputs a secret key sk and an evaluation key evk.
- $\mathsf{AuthAndEvalKey}(\mathsf{sk}) \to \{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}, L_o^{(0)}, L_o^{(1)}$: the authentication algorithm takes the secret key as input and outputs all input labels $\{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}$. There is a succinctness requirement that HMAC must satisfy: labels must have sizes bounded by $\mathsf{poly}(\lambda)$ where 1^λ is the security parameters.
- $\mathsf{EvalTag}(\mathsf{evk}, (\mathbf{x}, \{L_i^{(x_i)}\}_{i\in[\ell]})) \to L_o^{(b_o)}$: the evaluation-over-tag algorithm takes the evaluation key, the circuit, and the input $\mathbf{x} \in \{0,1\}^\ell$, it outputs $L_o^{(0)}$ if $C(\mathbf{x}) = 0$, or $L_o^{(1)}$ if $C(\mathbf{x}) = 1$.

One can construct PFGC as follows.

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{gk})$: the one-time setup algorithm runs $\mathsf{HMAC.KeyGen}(1^\lambda)$ to obtain the secret key sk and the evaluation key evk. It outputs $\mathsf{pp} = \mathsf{evk}$ and $\mathsf{gk} = \mathsf{sk}$. The sizes of pp and sk are bounded by $\mathsf{poly}(\lambda)$.
- $\mathsf{Garble}(\mathsf{gk}) \to \Gamma, \{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}, L_o^{(0)}, L_o^{(1)}$: the encoding algorithm parses $\mathsf{gk} = \mathsf{sk}$ and outputs $\Gamma = \epsilon$ and $\{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}, L_o^{(0)}, L_o^{(1)}$ from $\mathsf{HMAC.AuthAndEvalKey}(\mathsf{sk})$.
- $\mathsf{Evaluate}(\mathsf{pp}, \Gamma, (\mathbf{x}, \{L_i^{(x_i)}\}_{i\in[\ell]})) \to L_o^{(b_o)}$: the evaluation algorithm parses $\mathsf{pp} = \mathsf{evk}$ and runs $\mathsf{HMAC.EvalTag}(\mathsf{evk}, (\mathbf{x}, \{L_i^{(x_i)}\}_{i\in[\ell]}))$ to obtain $L_o^{(b_o)}$.

Succinctness. The scheme is succinct because $|\Gamma| = 0$ and the size of the labels in $\mathsf{Garble}(\mathsf{gk})$ does not grow linearly to $|C|$. Note that succinctness only considers the communication overhead, but not the compute overhead. The compute overheads of the garbler and the evaluator both grow linearly to $|C|$.

3.3 From AB-LFE

AB-LFE can reduce the compute overhead of the garbler to not grow linearly with $|C|$, but at most its depth. This dual succinctness to computation and communication overheads is often called "reusable" [GKP+13]. Specifically, AB-LFE from [HLL23] can make the compute overhead of the garbler independent of the depth of the circuit.

We present the detailed constructions in Appendix B and we keep only the high-level idea here.

AB-LFE. AB-LFE stands for attribute-based laconic function evaluation [QWW18]. We consider the following syntax for AB-LFE for a specific circuit $C : \{0,1\}^\ell \to \{0,1\}$, which departs from the syntax in [QWW18] by requiring the AB-LFE to provide all input labels at once.[10]

- $\mathsf{UrsGen}(1^\lambda, \mathsf{params}) \to \mathsf{urs}$: the URS generation algorithm takes the security parameters 1^λ and some other parameters as input and outputs a uniform reference staring (URS).
- $\mathsf{Enc}(\mathsf{urs}, \mu_0, \mu_1) \to \{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}, \mathsf{ct}_0, \mathsf{ct}_1$: the encryption algorithm takes the URS and secrets μ_0, μ_1 as input and outputs all input labels $\{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}$ and a ciphertext ct_0 and ct_1. This algorithm takes time that does not grow with $|C|$, but at most with its depth.
- $\mathsf{Dec}(\mathsf{urs}, (\mathbf{x}, \{L_i^{(x_i)}\}_{i\in[\ell]}), \mathsf{ct}_{b_o}) \to \mu_{b_o}$: the decryption algorithm takes the URS, the input $\mathbf{x} \in \{0,1\}^\ell$, the input labels $\{L_i^{(x_i)}\}_{i\in[\ell]}$, and the ciphertext ct_{b_o}, it outputs μ_{b_o} if $C(\mathbf{x}) = b_o$.

One can create a reusable garbling scheme as follows.

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{gk})$: the one-time setup algorithm runs $\mathsf{ABLFE.UrsGen}(1^\lambda, \mathsf{params})$ to obtain the URS. It outputs $\mathsf{pp} = \mathsf{urs}$ and $\mathsf{gk} = \mathsf{urs}$. Note that a URS can be compressed if it is generated from a PRNG seed. This allows $|\mathsf{pp}| = |\mathsf{gk}| = \lambda$.
- $\mathsf{Garble}(\mathsf{gk}) \to \Gamma, \{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}, L_o^{(0)}, L_o^{(1)}$: the encoding algorithm samples $L_o^{(0)} \leftarrow \{0,1\}^\lambda$ and $L_o^{(1)} \leftarrow \{0,1\}^\lambda$ and obtains the labels and ciphertexts by running $\mathsf{ABLFE.Enc}(\mathsf{urs}, \mu_0, \mu_1)$. It outputs $\Gamma = (\mathsf{ct}_0, \mathsf{ct}_1)$ and $\{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}$, $L_o^{(0)} = \mu_0$, and $L_o^{(1)} = \mu_1$.
- $\mathsf{Evaluate}(\mathsf{pp}, \Gamma, (\mathbf{x}, \{L_i^{(x_i)}\}_{i\in[\ell]})) \to L_o^{(b_o)}$: the evaluator algorithm parses $\mathsf{pp} = \mathsf{urs}$, $\Gamma = (\mathsf{ct}_0, \mathsf{ct}_1)$, runs $\mathsf{ABLFE.Dec}(\mathsf{urs}, (\mathbf{x}, \{L_i^{(x_i)}\}_{i\in[\ell]}), \mathsf{ct}_{b_o})$ to obtain μ_{b_o} and output $L_o^{(b_o)} = \mu_{b_o}$.

Succinctness and reusability. This scheme is succinct because $|\Gamma|$ does not grow with $|C|$ but at most its depth. The scheme is "reusable" because the computational overhead to generate a new GC does not grow with $|C|$ but at most its depth.

Concrete performance. AB-LFE is only nearly practical.

- AB-LFE with bounded depth [QWW18] requires that the parameters grow polynomially with the depth of the circuit. In BitVM, a circuit for a Groth16 verifier has a high depth.
- AB-LFE with unbounded depth [HLL23] solves the depth issue by removing "noise" in the middle of the computation, but there is an expensive cost to remove noise every time, and we expect that such noise removal needs to be performed very frequently.

[10] Note that not all AB-LFE schemes can provide all input labels at once, and not all of them provide the labels in a fully decomposable manner (such as [GKP+13]), but for the two constructions [HLL23; QWW18] which we consider, this is the case. As a result, we are using AB-LFE in a non-black-box manner.

4 Malicious security compilers

PFGC naturally provides malicious security against the evaluator, called "authenticity" discussed in § 2.1. In this section, we focus on malicious security against the garbler.

4.1 C&C

We can perform cut-and-choose (C&C) [Pin03; LP07], specifically SingleCut [Lin13; Bra13; HKE13; AMPR14; ZHKs16], to check if the garbler has correctly generated the circuit.

- **Step 1: generate and commit to s copies.** The protocol asks the garbler to generate s copies of the GC. When generating each GC, we assume that the garbler uses a pseudorandom generator (PRNG) with a random seed as the sole source of randomness. The garbler creates commitments for the seeds, hashes these circuits, publishes the commitments and the hashes.
- **Step 2: perform a public random coin toss.** A public random coin toss is performed,[11] which selects s_1 out of the s commitments that the garbler must reveal.
- **Step 3: verify the opened copies.** After revealing the PRNG seeds, the evaluator can rerun the garbling algorithm using the seed to recreate the same copy that matches the hash. The garbler passes C&C if all the revealed copies are correct.
- **Step 4: combine the remaining copies.** The remaining $s_2 = s - s_1$ copies are not opened. The garbler is asked to show these copies (not the seeds, but just the resulting GC), which are compared with the hashes. The new GC is a collection of these s_2 copies. The evaluator only needs to find one copy that is correct to evaluate.

Parameters. A simple SingleCut is to generate s copies, and for each copy, a public coin toss is made, and with $1/2$ probability, the copy is revealed and checked. This allows one to achieve a security level of 2^{-s}. In practice, choosing $s = 40$ is sufficient.

Integration with Bitcoin. The new GC in C&C comprises s_2 copies of the original GC. This increases the off-chain cost for the evaluator by s_2 times (storage overhead and worst-case compute overhead, but not the average compute overhead), which we cannot avoid. However, we want to make sure to that the on-chain cost is *not* increased by s_2 times, which is done as follows.

- The BitVM3 paper [Lin25b] suggests using a short key to encrypt s_2 input labels (for the same value and the same input wire, but in different copies) and instead of revealing all s_2 labels on-chain, the garbler reveals the short key instead. NIZK is used to ensure that the ciphertexts encrypted under the short key correspond to the input labels of each copy.

  This corresponds to the **Assert-i** transaction in standard BitVM and the **InputLabels-i, j** transaction in reverse BitVM.
- The evaluator only needs to submit a valid output label for *one* of the s_2 copies. We can construct a Taproot tree with s_2 leaves, each representing the script that accepts and verifies the output label, and the evaluator only needs to choose a leaf. This would slightly increase the overhead

[11] We can reuse Bitcoin proof-of-work (PoW) for this random coin toss, so that cut-and-choose can be performed in front of the public and only needs statistic security parameters. This is done by publishing these commitments on the Bitcoin blockchain and using the subsequent block header hashes to serve as the randomness for this coin toss.

of the on-chain cost due to the need to include $\log s_2 \approx 5$ hashes to select the Taproot leaf, but it is acceptable.

This corresponds to the **Disprove-i, j** transaction in standard BitVM and the **Prove-i, j** transaction in reverse BitVM.

Batched C&C. In the batched setting [LR14; HKK+14], one can improve C&C and reduce s. For example, if we want to generate 2^{10} PFGC at once, we can generate $s' = 7229$ copies, check about $s'_1 = 1085$ of them (each copy has a chance of 0.15 to be opened and checked), and randomly map the remaining ones into buckets of size 6, so that each of the original 2^{10} corresponds to six copies [LR14]. Eventually, the evaluator only needs to evaluate 6 copies each time, rather than ≈ 20.

However, doing C&C in large batches is not useful if the garbler and the evaluator are not expected to run the BitVM protocol that many times. In such a situation, one should consider DUPLO [KNR+17], following the idea of LEGO [NO09], which shows that if the GC consists of subcomponents that repeat many, many times, one can perform the C&C for the batched setting over these repeated elements and then solder them together. Many ZK proof systems that we consider have repeated subcomponents.

Advantage with free-XOR. We mentioned previously in § 3 that Minicrypt PFGC (or standard GC) has an advantage over HMAC and AB-LFE in that XOR gates remain free. This is also an advantage of C&C over NIZK and fraud proofs on Bitcoin because C&C retains free-XOR, but:

- **NIZK.** Although proving XOR gates can be made significantly cheaper than proving AND gates (note that AND gates involve hash functions), there is still a cost. To prove XOR gates in a GC, one needs to prove the "byte-by-byte" XOR of the labels,[12] requiring bit decomposition, lookup tables, and logUp arguments [Hab22], in most prime order NIZK schemes. Binius [DP24; DP25] may allow XOR to be almost free (only the logUp arguments), which we will discuss later.
- **Fraud proof on Bitcoin.** For fraud proof on Bitcoin, although the fraud proof for an XOR gate can be made significantly cheaper as it does not involve a hash function like AES, bit decomposition and lookup tables for XOR are needed, so there is still a cost, and it would make the Taproot tree somewhat larger.

How useful free-XOR is depends on the proportion of XOR gates in the circuit. For a pairing-based SNARK over a prime field, the latest number from Citrea [Cit25] shows that 7.7 billion gates out of the 10.4 billion gates are free (about 75%). In this case, not having free-XOR increases the overhead by a factor of four, which might still be acceptable. For DV-SNARK over a binary curve as in Glock [Eag25], 3.27 billion gates out of the 3.29 billion gates are free (about 99.6%). In this case, not having free-XOR increases the overhead by about 250×, which might not be worthy of considering other schemes.

[12] One may wonder if XOR can be replaced by additions over characteristic larger than 2 (which would be more natural to work with the prime order field in common NIZK), but XOR is in fact a very unique operation that separates itself from additions. For example, assume a global key Δ as in free-XOR, we are looking for an operation $\star$ such that $(A \star \Delta) \star (B \star \Delta) = A \star B$. Additions satisfy this only when it is over a field with characteristic 2, implying bit decomposition, unless the NIZK's native field is of characteristic 2.

4.2 NIZK

We can also ask the garbler to generate a NIZK proof to show that the GC has been generated correctly, without revealing the specific labels. Since the NIZK proof is usually fairly small, it has better communication efficiency than C&C.

The Babylon team studies an idea using a ZK-friendly hash function, such as Poseidon2 [GKS23], to instantiate [ZRE15], where the garbler makes 2 calls to Poseidon2 for each AND gate. Poseidon2 can be proven very efficiently.

- StarkWare has a benchmark on S-two [Sta24] showing that S-two can prove 6×10^5 Poseidon2 per second on a single 12-core M3 Pro chipset. This translates to 3×10^5 AND gates per second.
  - If we have a circuit with 2.7×10^9 AND gates, it takes 2.5 hours with a single laptop.
  - If we have a circuit with 1.1×10^7 AND gates, it takes 36 seconds with a single laptop.
- Polyhedra has a benchmark on Expander [Pol24] showing that Expander can prove $\approx 1 \times 10^6$ Poseidon2 per second also on M3 Pro. This translates to 5×10^5 AND gates per second.
  - If we have a circuit with 2.7×10^9 AND gates, it takes 1.5 hours with a single laptop.
  - If we have a circuit with 1.1×10^7 AND gates, it takes 22 seconds with a single laptop.

This can be distributed to multiple machines or potentially with GPU. XOR gates also need to be proven, so they are no longer "free". This can become an issue when there are significantly more XOR gates than AND gates, as is the case with Glock [Eag25] when using a binary curve.

Changing to Poseidon2 has a cost in garbling and evaluating, since Poseidon2 is much slower than AES. But since AES is very efficient, we believe that computation is not the bottleneck, but memory performance (and disk performance for persistent storage of the GC). So, changing from AES to a slower, ZK-friendly one might not noticeably slow down the overall system.

One can consider ZK-friendly hash functions that are also optimized for native performance, such as the Monolith hash function [GKL+25]. With parameters tailored for garbled circuits, it should be only 2 to 4 times slower than SHA-256.

Alpen Labs is the first to study Binius [DP25; DP24] for proving GCs. Binius represents a family of NIZK on binary towers. It can efficiently prove AES [DR02] and Grøstl [GKM+21] (which is closely related to AES) when the underlying Binius binary towers use the AES tower (the field $\mathsf{GF}(2^8)$ with irreducible polynomial $x^8 + x^4 + x^3 + x + 1$). Another hash function that works well with Binius is the Vision Mark-32 hash function [AMPŠ24], which has competitive performance compared with Grøstl on FPGA, and is defined on the same binary tower $\mathsf{GF}(2^{32})$ that Binius uses. An advantage of Vision Mark-32 over Grøstl is that its S-box can be easily arithmetized and does not need a lookup table.

There are other NIZK algorithms, based on linear-time codes, that we can consider. They tend to have an *extremely* fast prover, but the proofs are somewhat long. However, such a large proof is incomparable to the size of the GC it is proving, so the proof sizes are really not an issue. There are several constructions:

- from Spielman's codes [Spi95]: Brakedown [GLS+23] and Orion [XZS22; DP23; HS24]
- from expand-accumulate (EA) codes [BCG+22]: a field-agnostic version of Brakedown [BFK+24]
- from repeat-accumulate-accumulate (RAA) codes [DJM98; GM08]: Blaze [BCF+25]

4.3 Fraud proofs on Bitcoin

An idea from Delbrag [Rub25a; Rub25b] that was recently revisited by Robin Linus in BitVM3s [Lin25a] with several practical instantiations (AES [DR02], SPECK-128 [BSS+15], PRINCEv2 [BEK+20]) is to allow for on-chain fraud proofs if at least one gate in the GC was not correctly garbled. This works for Minicrypt PFGC based on a fixed-key cipher or a hash function, but it does not apply to BitVM from AB-LFE or HMAC. It does not seem to work well with one-hot garbling [HK21]. XOR gates that were usually free [KS08] are not free anymore. To enable fraud proofs, each gate needs to include two more BitHash hashes. Computing the BitHash hashes and constructing the Taproot tree would take significantly more time than the original garbling, but still only involve hash functions.

The protocol works as follows.

The garbler needs to present two hashes of the labels for each wire of the GC. It is important that this hash function is not the one used in the garbling scheme itself, as it might lead to certain attacks, so we use H' here. For any wire w, the garbler shares $h_w^{(0)}$ and $h_w^{(1)}$:

$$h_w^{(0)} = H'(L_w^{(0)}) \qquad h_w^{(1)} = H'(L_w^{(1)})$$

The need to transmit two hashes for the output wire of each gate—XOR or AND—is why XOR gates are no longer free.

We also include a NIZK proof showing that the hashes for the input labels are consistent with the other hashes used in the **Assert-i** transaction in standard BitVM and the **InputLabels-i, j** transaction in reverse BitVM. The compute overhead only grows with the number of input bits.

A new Taproot tree is built for disproving the correct garbling, which includes a spending path for each gate g that is used to challenge the garbling of that gate. Consider a gate with input wires w_a and w_b and output wire w_c. The spending path for g is hardcoded with $h_{w_a}^{(0)}, h_{w_a}^{(1)}, h_{w_b}^{(0)}, h_{w_b}^{(1)}, h_{w_c}^{(0)}, h_{w_c}^{(1)}$ and requires the challenger to present $b_a \in \{0,1\}, b_b \in \{0,1\}, L_{w_a}^{(b_a)}, L_{w_b}^{(b_b)}$ that:

$$\begin{aligned} h_{w_a}^{(b_a)} &= H'(L_{w_a}^{(b_a)}) \\ h_{w_b}^{(b_b)} &= H'(L_{w_b}^{(b_b)}) \\ h_{w_c}^{(g(b_a,b_b))} &\neq H'(L_{w_c}^{g(b_a,b_b)}) \end{aligned}$$

where $L_{w_c}^{(g(b_a,b_b))}$ is the label computed in the Bitcoin script (i.e., which would incur gas costs) from evaluating the garbled gate on $L_{w_a}^{(b_a)}$ and $L_{w_b}^{(b_b)}$, usually involving the hash function H that is used in the garbling scheme. Now we discuss how to implement H and H' that can be verified on-chain efficiently and securely.

Implementing H. The most standard approach is to build H using AES [DR02] following [GKWY20; GKW+20]. The BitVM3s paper [Lin25a] estimates that the Bitcoin script size for AES is about 20 KB.

If we are okay to change this hash function used in garbling, the BitVM3s paper [Lin25a] also suggests other hash functions and block ciphers that are likely Bitcoin friendly, with candidates

Algorithm 1: BitHash [Lin25a]

- set v_0 to be a publicly known starting value (it is recommend using a pseudorandom value that is *instance*-specific, *wire*-specific, and *value*-specific that would not be repeated)
- for i from 0 to $L-1$:
  - if the i-th bit is 0, do $v_{i+1} = \mathsf{SHA\text{-}256}(v_i)$
  - otherwise, do $v_{i+1} = \mathsf{RIPEMD\text{-}160}(v_i)$
- output $\mathsf{RIPEMD\text{-}160}(v_L)$

like SPECK-128 [BSS+15] and PRINCEv2 [BEK+20]. There is a preference to look at block ciphers rather than hash functions, since hash functions generally have a larger input size and a larger output size, but GC does not need such a large input and output size.

Implementing H'. For instantiating H' for committing the hashes of input labels, [Lin25a] presents a new algorithm called BitHash, which is a Bitcoin-friendly fixed-length hash function $\{0,1\}^L \to \{0,1\}^{160}$ built from SHA-256 and RIPEMD-160, shown in Algorithm 1.

Comparison with C&C. We now compare fraud proofs on Bitcoin with C&C in terms of (1) computation overhead and (2) communication and storage overheads.

♦ **Computation overhead.** In C&C, if the operators and challengers are honest, there is no need to evaluate the GC. But if any operator or challenger is malicious, in the worst case, the evaluator (challenger in standard BitVM, operator in reverse BitVM) must evaluate each of the remaining s_2 GCs to find one GC that is correct. However, it is not easy for a malicious garbler to intentionally cause the worst-case overhead. For example, assume that $s = 40$ and $s_1 = s_2 = 20$.

- The best strategy for a malicious garbler to pass the C&C but ensure that the first randomly chosen GC among the s_2 remaining ones is not going to work, which is to only make 1 out of the $s = 40$ GCs incorrect. The probability is as follows.

$$\frac{\binom{39}{19}}{\binom{40}{20}} \cdot \frac{1}{20} = 2.5\%$$

- If we want to first *two* randomly chosen GCs to be not working, then the best strategy is to make 3 out of the $s = 40$ GCs incorrect. The probability is as follows.

$$\frac{\binom{37}{17}}{\binom{40}{20}} \cdot \frac{3}{20} \cdot \frac{2}{19} = 0.18\%$$

To summarize, for C&C, we can usually assume that the number of GCs to be evaluated by the evaluator would be one or two.

In fraud proofs on Bitcoin, there is only one GC, and this is the only GC to evaluate. However, the issue is that, before Alice sends money to the BitVM instance, the challengers need to agree on the Taproot tree that represents the fraud proofs on Bitcoin, the challengers need to generate

this Taproot tree on their own. Note that this Taproot tree has one leaf for each gate (XOR or AND), and the leaf is a hash of the script, which could be several KBs to tens of KBs each. The computation overhead for challengers before peg-in would be dominated by the computation of the Taproot tree, and it depends on the number of XOR and AND gates (the cost of an XOR gate differs from that of an AND gate). As we previously mentioned, a pairing-based SNARK verifier contains 75% XOR gates, and a DV-SNARK verifier contains 99.6% XOR gates. For DV-SNARK, fraud proofs on Bitcoin would cause all these XOR gates to become non-free.

A similar overhead will also appear when an evaluator actually wants to challenge a gate in the GC (after the input labels are revealed) as the transaction spending that Taproot UTXO will need to show a Merkle tree proof about the Taproot tree. We anticipate that evaluators will store the partial results from the previous computation of the Taproot tree before peg-in (which can be made much smaller than the GC) and use these partial results to avoid a full recomputation of the Taproot tree. This cost can be made much smaller than the other computation overhead involved in the challenge process, and for simplicity, we can ignore this cost.

To summarize, for the overall computation overhead, fraud proofs on Bitcoin are expected to be much more expensive due to Taproot tree. If we focus specifically on the evaluation of the GC, then the computation overhead should be similar.

♦ **Communication and storage overheads.** In C&C, each AND gate requires $128 \cdot s_2$ bits, where s_2 is the number of unopened GCs, but the XOR gates are free. Standard C&C gives $s_2 \approx 20$, while batched C&C or DUPLO [KNR+17] can lower to $s_2 \approx 6$. The GC needs to be downloaded and stored by the evaluators (challengers in standard BitVM, operators in reverse BitVM). The blowup of the communication and storage overheads is s_2 times.

For fraud proofs on Bitcoin, the blowup depends on the portion of free gates (i.e., XOR gates) in the GC, as these free gates are no longer free in terms of communication and storage. For each gate, we need to store the hashes of both labels of the output wire, so 320 bits per gate (using H' with 160-bit output). This is in addition to the original 128 bits per AND gate for the Minicrypt PFGC. As we mentioned previously, a pairing-based SNARK verifier contains 75% XOR gates. The blowup of the communication and storage overhead is as follows.

$$\frac{320 + 0.25 \cdot 128}{0.75 \cdot 0 + 0.25 \cdot 128} = 11$$

This would be smaller than standard C&C but bigger than batched C&C or DUPLO.

If we consider a DV-SNARK verifier, which contains 99.6% XOR gates. The blowup is as follows.

$$\frac{320 + 0.004 \cdot 128}{0.996 \cdot 0 + 0.004 \cdot 128} = 626$$

In summary, fraud proofs on Bitcoin for pairing-based SNARK can beat standard C&C, but not batched C&C or DUPLO. And fraud proofs on Bitcoin would be too heavy for the DV-SNARK verifier due to the fact that XOR gates are no longer free in fraud proofs for Bitcoin.

5 Proof systems

In this section, we discuss various proof systems that may be used for BitVM. We want to minimize the on-chain and off-chain verifier overheads.

- on-chain cost depends on the proof size
- off-chain cost depends on the size of the Boolean circuit that computes the verifier

Publicly verifiable SNARK. The situation with a publicly verifiable proof system is straightforward. The one proof system that offers the shortest proofs and the fewest bilinear pairings is Eagen25 [Eag25], which offers a proof of 762 bits with 2 bilinear pairings for verification. There are many criteria in selecting secure pairing-friendly curves [BD19; BL25]. In practice, we use some common and standardized curves, such as BN254. For STARK, we defer the discussion to Appendix C.

DV-SNARK. Liam Eagen [Eag25] from Alpen Labs is the first to consider designated-verifier SNARK (DV-SNARK) [GKR08] for BitVM for smaller proof sizes and lower compute overhead for the verifier. Unlike publicly verifiable SNARK, the DV-SNARK proofs can only be verified by designated verifiers, which are parties who know the verifier secrets. It is important that the prover does not know such verifier secrets. Otherwise, they may enable the prover to forge proofs.

When there are many challengers, we assume that each challenger will have their own different verifier secrets, and therefore each of them would expect a different proof.

For standard BitVM, two changes are needed.

- The operator needs to generate a different GC for each pair of the challenger. Let N be the number of challengers. This would increase the overheads for the operators:
  - The wall clock time for each operator grows from $O_{\lambda,\kappa}(|C|)$ to $O_{\lambda,\kappa}(N \cdot |C|)$.
  - The total compute overhead grows from $O_{\lambda,\kappa}(M \cdot |C|)$ to $O_{\lambda,\kappa}(MN \cdot |C|)$.
  - The storage overhead for each operator grows from $O_{\lambda,\kappa}(|\mathsf{in}|)$ to $O_{\lambda,\kappa}(N \cdot |\mathsf{in}|)$, in total from $O_{\lambda,\kappa}(M \cdot |\mathsf{in}|)$ to $O_{\lambda,\kappa}(NM \cdot |\mathsf{in}|)$.

  The transaction flow needs to be modified to accommodate multiple challengers challenging the same claim, which would borrow some techniques from reverse BitVM. In fact, the resulting protocol would be very similar to reverse BitVM except that the evaluators are still the challengers. The performance of standard BitVM now is basically the same as that of reverse BitVM.
- For challengers to be able to disprove, an additional step is needed for challengers to obtain the input labels for the DV-SNARK verification key using maliciously secure OT [NP01; CO15], which allows challengers to obtain the input labels without revealing the verification key to operators. The challengers should verify the correctness of the input labels.

For reverse BitVM, the operators need to obtain the input labels for the DV-SNARK verification key from the challengers who generate the GCs (note that each challenger has a different GC for each operator, corresponding to a different DV-SNARK parameters). The operators will need to verify a NIZK proof showing that the requested input labels correspond to the verification keys of that DV-SNARK with the prescribed SRS parameters.

DV-SNARK requires that the SRS be generated by the designated verifiers—which needs to be verified. There is a cost to prove and verify that the part of the SRS generated by the designated verifiers is correct. This cost is necessary because an incorrect SRS may trivially prevent an honest operator from generating a valid proof and claiming the deposited BTC following the protocol. We will soon see that this cost plays an important role in choosing DV-SNARK proof systems.

There are several families of DV-SNARK protocols based on different compilers that are used to construct the SNARK.

- **Orrù25 compiler.** [Orr25] showed that one can modify the KZG commitment scheme [KZG10] into a designated-verifier variant, which we call DV-KZG, that does not need the verifier to do any pairing. Orrù also showed an IOP compiler that can compile any polynomial IOP (e.g. Plonk, Pari) with DV-KZG into a DV-SNARK. Orrù25 compiler can be generalized to equifficient polynomial commitments (EPC) [DMS24].
- **BCIOP13 compiler.** Bitansky, Chiesa, Ishai, Ostrovsky, and Paneth [BCI+13] showed how to compile a linear interactive proof with a linear-only encryption into a DV-SNARK.
- **BHIRW24 compiler.** Bitansky, Harsha, Ishai, Rothblum, and Wu [BHI+24] showed how to compile a linear interactive proof into a DV-SNARK in the generic group model [Sho97], but the CRS is one-time, which implies additional communication and computation overheads, and each time the CRS needs to be proven to be valid.

We now discuss each of these options. But our conclusion is that Orrù25 compiler, which Alpen Labs first researched and adopted, appears to be most suitable for BitVM.

5.1 Orrù25 compiler

Many SNARK protocols [CHM+20; GWC19], are based on KZG polynomial commitment [KZG10]. One can replace that polynomial commitment with Orrù's designated-verifier KZG [Orr25], as follows, to remove pairings. This can significantly reduce the size of the GC of the SNARK verifier, though the SNARK now is designated-verified, rather than publicly verifiable.

- $\mathsf{Gen}(1^\lambda, \ell) \to \mathsf{srs}, \mathsf{st}$: Using a well-known group generator G, it samples a random number τ and computes:
$$G_i = \tau^i \cdot G \qquad \text{for} \quad i \in [\ell]$$
It outputs the structured reference string $\mathsf{srs} = \{G_i\}_{i\in[\ell]}$ and the verifier's state $\mathsf{st} = \tau$.
- $\mathsf{Commit}(\mathsf{srs}, f) \to \mathsf{cm}$: Given the coefficients of the polynomial $f(X)$, it computes and outputs the commitment as:
$$\mathsf{cm} = \sum_{i=0}^{\ell-1} f_i \cdot G_i$$
- $\mathsf{Open}(\mathsf{srs}, f, x) \to \pi$: Compute $g(X) = \frac{f(X)-f(x)}{X-x}$. It computes and outputs the proof π as:
$$\pi = \sum_{i=0}^{\ell-2} g_i \cdot G_i$$

- $\mathsf{Verify}(\mathsf{st}, \mathsf{cm}, x, y, \pi)$: It is easy to verify that $f(x) = y$ by checking that:

$$(\tau - x) \cdot \pi = \mathsf{cm} - y \cdot G$$

Verify the SRS. It is easy to verify the SRS using a random linear combination and Chaum-Pedersen protocol [CP92]. The Chaum-Pedersen protocol, given group elements (A, B, C) in the same group where $A = a \cdot G$, $B = b \cdot H$, $C = c \cdot H$ where G does not have to be H as long as they have the same orders, can prove $c = ab$ without revealing a, b, or c. The protocol can be made non-interactive, which works as follows:

- The prover samples a random scalar r of the group and sends $A' = r \cdot G$ and $C' = r \cdot B$.
- The prover uses Fiat-Shamir transform to compute a challenge $t = H(A \parallel B \parallel C \parallel A' \parallel C')$.
- The prover sends $s = r + ta$.
- A verifier can check the Fiat-Shamir transform and check if $s \cdot G = A' + t \cdot A$ and $s \cdot B = C' + t \cdot C$. If so, we know that $C = a \cdot B = ab \cdot H$.

We can prove the entire SRS using a single invocation of the Chaum-Pedersen protocol. Generate a random number α. Compute:

$$A = \sum_{i=1}^{l-1} \alpha^i \cdot G_i \qquad B = G_1 \qquad C = \sum_{i=1}^{l-1} \alpha^i \cdot G_{i+1}$$

The DeMillo-Lipton-Schwartz–Zippel lemma [DL78; Zip79; Sch80] shows that, let $G_1 = \tau \cdot G$, as long as the scalar field is large enough, with an overwhelming probability,

$$\tau \cdot G_i = G_{i+1}$$

which implies $G_i = \tau^i \cdot G$ by induction. Alternatively, if the curve is pairing-friendly, one can check this relationship with pairing given a G_2 element with scalar τ (this is commonly used when the SRS is generated from a setup ceremony where nobody knows τ).

Curve selection. With the modification from KZG to DV KZG in the Orrù compiler, we no longer need the elliptic curve to be pairing-friendly. Instead, binary curves such as K-233 or K-283—an insight from Alpen Labs—can be implemented efficiently in GC [Por22; Por23]. Specifically, results show that the number of non-free gates (AND gates) in the GC can be reduced to about 11 million, and most of the other gates (99.6% of them) are XOR gates, which are free.

Generalization to EPC. The Orrù compiler naturally works for the construction of equifficient polynomial commitment (EPC) in [DMS24]. Let $\mathbf{\Lambda}$ be the set of equifficient constraints, $\mathbf{\Theta}$ be the set of non-trivial constraints, and let W be the first non-trivial constraint. The EPC verifier checks, for W and all trivial constraints $V \in \mathbf{\Lambda} \setminus \mathbf{\Theta}$:

$$e(c_W, \delta_W \cdot H) = e\left(\left\langle \frac{\boldsymbol{\alpha}_W}{\gamma_W} G, \boldsymbol{v}_W \right\rangle + \sum_{V \in \mathbf{\Lambda} \setminus \mathbf{\Theta}} \left\langle \frac{\boldsymbol{\alpha}_V \cdot G}{\gamma_W}, \boldsymbol{v}_V \right\rangle, \gamma_W \cdot H\right) \cdot e(\mathsf{w}_W, \gamma_W \tau \cdot H - \gamma_W u \cdot H)$$

where c_W is a commitment, w_W is an opening proof, u is the evaluation point, and $\boldsymbol{\alpha}_W$, δ_W, γ_W, and τ are secrets in the SRS that a verifier will know in the DV-SNARK setting. It is straightforward to see that, a DV-SNARK verifier who knows $\boldsymbol{\alpha}_W = (1, \alpha_W, \alpha_W^2, ...)$, δ_W, γ_W, and τ can instead check the following equation without pairings:

$$\frac{\delta_W}{\gamma_W} \cdot c_W = (\tau - u) \cdot \mathsf{w}_W + \left(\left\langle \frac{\boldsymbol{\alpha}_W}{\gamma_W}, \boldsymbol{v}_W \right\rangle + \sum_{V \in \boldsymbol{\Lambda} \setminus \boldsymbol{\Theta}} \left\langle \frac{\boldsymbol{\alpha}_V}{\gamma_W}, \boldsymbol{v}_V \right\rangle \right) \cdot G$$

And for each remaining non-trivial constraint $S \in \boldsymbol{\Lambda} \cap \boldsymbol{\Theta} \setminus \{W\}$,

$$e(c_S, \delta_S \cdot H) = e\left(\left\langle \frac{\boldsymbol{\alpha}_S}{\gamma_S} G, \boldsymbol{v}_S \right\rangle, \gamma_S \cdot H \right) \cdot e(\mathsf{w}_S, \gamma_S \tau \cdot H - \gamma_S u \cdot H)$$

This can be similarly rewritten as follows given that the DV-SNARK verifier knows $\boldsymbol{\alpha}_S$, δ_S, γ_S, and τ:

$$\frac{\delta_S}{\gamma_S} \cdot c_S = (\tau - u) \cdot \mathsf{w}_S + \left\langle \frac{\boldsymbol{\alpha}_S}{\gamma_S}, \boldsymbol{v}_S \right\rangle \cdot G$$

By applying this compiler to Polymath, Pari, and Eagen25, one can obtain the DV-SNARK versions that remove pairings and allow for a more flexible selection of curves.

Verify EPC SRS. An additional technicality is that EPC has an additional SRS, which means that the operators will need to verify this additional SRS.

In the construction of the EPC in [DMS24], the additional SRS consists of a committer key, an opener key, and a verifier key, all of which need to be verified. It is generated by a special role called *specializer*, which would be the challengers in BitVM.

For each non-trivial constraint $S \in \boldsymbol{\Lambda} \cap \boldsymbol{\Theta}$,

- The committer key includes:

  $$\mathsf{ck}_S = \frac{\sum_{i=1}^{|S|} \alpha_S^{i-1} \cdot \boldsymbol{b}_S^{(i)}}{\delta_S}$$

  where $\boldsymbol{b}_S^{(i)}$ is a vector of group elements that serve as the generators of the basis for that non-trivial constraint. This vector can be computed from the KZG SRS without knowing the secret τ, by taking a linear combination of the KZG SRS.
- The opener key includes:

  $$\mathsf{ok}_S = \left(\frac{1}{\gamma_S} \cdot \boldsymbol{\Sigma}, \frac{\alpha_S}{\gamma_S} \cdot \boldsymbol{\Sigma}, ... \right)$$

  where $\boldsymbol{\Sigma} = \mathsf{srs}$ is the vector of group elements that represent the canonical monomial basis that equals to the KZG SRS.
- The verifier key includes:

  $$\mathsf{vk}_S = \left(\frac{\boldsymbol{\alpha}_S \cdot G}{\gamma_S}, \gamma_S \cdot H, \gamma_S \tau \cdot H, \delta_S \cdot H \right)$$

  where H is the generator over the other group $\mathbb{G}_2$ for the pairing-friendly curve.

To prove that the committer key is generated correctly, the specializer generates a random element $R_1 = r \cdot G$ and computes $R_2 = r\gamma_S \cdot G$, $R_3 = \frac{r\gamma_S}{\delta_S} \cdot G$, and $R_{4,i} = \frac{r\alpha_S^i}{\delta_S} \cdot G$. The Chaum-Pedersen protocol can verify that $R_2, R_3, R_{4,i}$ is consistent with vk_S. Then, the specializer uses Fiat-Shamir transform to take a random linear combination of $\boldsymbol{b}_S^{(i)}$ and denotes the result as $R_{5,i}$. It computes $R_{6,i} = \frac{r\alpha_S^i}{\delta_S} \cdot R_{5,i}$ and proves this relation using the Chaum-Pedersen protocol with the help of $R_{4,i}$. Note that $\sum_{i=1}^{|S|} R_{6,i} = r \cdot \mathsf{ck}_S$, which can be proven using the Chaum-Pedersen protocol with the help of R_1. The same protocol can be used to verify the consistency between ok_S and vk_S.

For each trivial constraint $V \in \boldsymbol{\Lambda} \setminus \boldsymbol{\Theta}$,

- The committer key includes:
$$\mathsf{ck}_V = \frac{\alpha_V}{\delta_W} \cdot \boldsymbol{\Sigma}$$
where W is the first non-trivial constraint.
- The opener key includes:
$$\mathsf{ok}_V = \frac{\alpha_V}{\gamma_W} \cdot \boldsymbol{\Sigma}$$
- The verifier key includes:
$$\mathsf{vk}_V = \frac{\alpha_V \cdot G}{\gamma_W}$$

It can be proven in the same manner relying on $(\gamma_W \cdot H, \gamma_W \tau \cdot H, \delta_W \cdot H)$ in vk_W.

Passing input labels for verifier key. The GC requires the verifier key as input.

- For standard BitVM, the garblers are the operators, and the evaluators are the challengers, who are the designated verifiers that know the DV-SNARK verifier key.
- For reverse BitVM, the garblers are the challengers, and the evaluators are the operators, who will evaluate the GC but cannot know the DV-SNARK verifier key.

The evaluator must obtain the input labels for the verifier key without leaking the verifier key to the garbler, and the garbler needs to verify that the input labels requested by the evaluator correspond to the SRS. To do that, the evaluator uses maliciously secure oblivious transfer (OT) to obtain the input labels. The labels are checked against the commitments of these input labels made previously in cut-and-choose or NIZK for the correct garbling of this garbled circuit.

In addition, a separate NIZK is needed to show that the requested labels correspond to the SRS. This can be done by combining a NIZK and the Schnorr protocol [Sch89]. See [SSS+22; KMN23; WOS+25] for how to construct such NIZK efficiently.

5.2 BCIOP13 compiler

Another compiler for creating DV-SNARK is the BCIOP13 compiler [BCI+13]. It shows that one can construct DV-SNARK by combining an input-oblivious[13], two-message linear interactive proof (LIP) and linear-only homomorphic encryption. For the background, this class of interactive proof systems restricts the behaviors of the prover and the verifier:

[13]Also known as "instance-oblivious".

- The protocol is **two-message**, which restricts the protocol to start with a message from the verifier (called *query*), followed by a message from the prover (called *answer*), and nothing more.
- The protocol is **input-oblivious**, meaning that the query (first message) that the verifier sends do not depend on the input.
- The protocol is **linear**, meaning that the answer (second message) that the *honest* prover sends is simply the result of applying a *linear* function over the elements in the query (viewed as a vector) from the verifier. This coefficients of the linear function can depend on the input, the witness, or the current round number.

Such an input-oblivious, two-message linear interactive proof protocol can be used to create a DV-SNARK using the BCIOP13 compiler.

- Instead of sending the query in plaintext, the verifier encrypts each element of the query with the linear-only homomorphic encryption and sends the encrypted query. Only the verifier has the decryption key.
- The prover applies the linear function over the encrypted query and sends the result as the answer. This is possible because the encryption is homomorphic to linear functions and can be performed without knowing the decryption key.
- To verify the proof, the verifier decrypts the answer and runs the verification algorithm of the original LIP protocol.

Linear-only homomorphic encryption. There are several cryptosystems to consider: Paillier [Pai99; DJ01], ElGamal [ElG84], Benaloh [Ben84], and lattice-based encryption [Reg05; BV11; GMNO18], all under some linear-only assumptions. We want to discuss the limitations of each of these cryptosystems, which is also a reflection of the limitations of the BCIOP13 compiler.

Limitations for ElGamal and Benaloh. An issue with ElGamal and Benaloh is that decryption requires some sort of discrete log, and in order to make this discrete log practical, the subgroup size cannot be too large. This issue fundamentally leads to the following limitations.

- Due to the smaller subgroup size, the soundness is often limited and needs *soundness amplification*, which means that the same proof needs to be generated a few times, leading to a longer proof, although the designated challenger in BitVM only needs to challenge one of them.
- The discrete log is still very expensive to perform inside GC, and a way to make it work for BitVM is to enable the challenger to compute the discrete log off-chain and then request the input labels for the discrete log result through an interaction with the garbler, which can be on-chain or off-chain. This discrete log is verified rather than computed in GC. This approach has a *fundamental* caveat that the challenger would not be able to compute the discrete log if the prover was malicious, and it is difficult for the GC to verify whether the proof has been honestly generated in a way that discrete log is easy.
- In many DV constructions, in order to make discrete log practical, the program must be formatted as a Boolean circuit, which is often much more expensive than an arithmetic circuit.

There have been recent work on improving ElGamal-based DV-SNARK from the BCIOP13 compiler, but we find that the fundamental technical challenge still remains difficult. We provide a detailed discussion in Appendix D.

Limitations for Paillier. Paillier has the problem that decryption is slow, which causes the verifier to be slow. The decryption requires performing modular exponentiations modulo N^2 where $N = pq$ is a suitable RSA modulus of about 2048 bits, so this is modulo a 4096-bit number, all within a GC. Moreover, this is not multiplication, but exponentiation, and the exponent is a 2048-bit number $\phi(N) = (p-1)(q-1)$.

Traditionally, if the verifier knows p and q, the verifier can use the Chinese Remainder Theorem (CRT) and other computation tricks that we outlined in Appendix A.1 to help with the computation, but this is not really suitable for GC because the topology of the GC, including gate types and connections, must not depend on p and q, which makes much of such optimization not useful. There are GC schemes that can hide the topology, either through a universal circuit [Val76] (with many subsequent works) or through multiparty computation [KM11; MS13], but at a very high cost and often with new restrictions.

Limitations for lattice-based encryption. If we conjecture that some lattice-based cryptosystem is *linear-only*,[14] we can also instantiate the BCIOP13 compiler with lattice-based encryption. An important advantage is that lattice-based encryption often has very efficient decryption, which avoids the major limitations of ElGamal and Benaloh and of Paillier. A unique advantage is that lattice-based DV-SNARK may likely offer post-quantum security. Note that all instantiations from the Orrù25 compiler and the BHIRW24 compiler are not post-quantum secure because they rely on discrete log being hard on certain groups.

Several recent constructions of lattices [ISW21; SSE+24] have very competitive performance for the verifiers, but the proof size can go up to around 8KB (which increases the on-chain cost). The verifier can be much more efficient than publicly verifier SNARK such as Groth16 because pairings require elliptic curves and the field must be large, while these lattice-based primitives can use small fields. Specifically for [ISW21], the proof size and the overhead of the verifier can be further reduced if zero knowledge is not needed (which is the BitVM setting). If an application has to require zero knowledge, one can bypass this restriction by recursive proof composition [BCCT13].

One caveat to pay attention to is that it does not guarantee *reusable soundness*. It is okay to reuse the same SRS as long as the operator submits a valid proof. Once the operator starts to submit invalid proofs, a new SRS should be generated.

Note that both schemes use *soundness amplification* to achieve sufficient security, which is to repeat the same proof many times, each time under a different SRS. It is expected that if a computation is invalid, one of those SRSs would discover that. This soundness amplification is actually compatible with standard BitVM (but not reverse BitVM), as the challenger only needs to run the verifier on the GC for one of these proofs, under a specific SRS, that fails the verification.

Lattice-based DV-SNARKs [ISW21; SSE+24] appear to be the most practical under the BCIOP13 compiler. If we compare them with DV Eagen25 under the Orrù25 compiler, we know that the proof size of DV Eagen25 is 10 to 20 times smaller than that of lattice-based DV-SNARKs, which reduces the on-chain cost. If DV Eagen25 is instantiated with binary curves, we believe that

[14] This conjecture is debatable. The BCIOP13 paper [BCI+13] does not consider lattice-based cryptosystems because they feel less confident in their linear-only property since lattice cryptography has been used to instantiate fully homomorphic encryption.

DV Eagen25 may be more efficient than lattice-based DV-SNARKs. This comparison ignores the differences in the reusability restrictions and verifiability.

Verify the SRS. One important caveat of the BCIOP13 compiler constructions is that it can be very difficult to verify the SRS, depending on the underlying LIP scheme. Although generic NIZK can always be used to verify any SRS, it could be very computationally expensive. The approach we use for the Orrù25 compiler works for certain linear PCP constructions [GGPR13; BSCG+13] and for certain cryptosystems that allow us to take a random linear combination and sum up multiple elements together. This would work for Paillier, ElGamal, and Benaloh, but not for lattice-based cryptosystems due to noise. A proof is needed to show that two ciphertexts are encrypting the same data. For ElGamal and Benaloh, this can be done using the Chaum-Pedersen protocol or the Schnorr protocol. For Paillier, a similar approach would work, with the caveat that a random linear combination can be pretty heavy since Paillier uses a large modulus.

5.3 BHIRW24 compiler

Another compiler that is closely related to the BCIOP13 compiler is from [BHI+24]. The focus of [BHI+24] is to construct very short proofs for dot-product proof (DPP), which refers to 1-query fully linear PCP, and then [BHI+24] presents a compiler in Section 7.2 that can compile DPP to DV-SNARK with the limitation that it requires prover-verifier interaction and non-reusable SRS.

We generalize this idea to Groth16. One can think of Groth16 as the more aggressive version of the BCIOP13 compiler in which each field element of the answer is only compiled into one group element (instead of two), at the cost that security can only be proven under the generic group model (GGM) [Sho97]. There are two proof systems given in Groth16, one with 3-element (which is what we usually refer to), and one with 2-element. They can be compiled into different DV Groth16 systems, with different proof sizes, as follows.

For brevity, we remove the zero-knowledge property of Groth16 here.

DV (3-element) Groth16. In Groth16, a relation is defined between public input $(a_1, a_2, ..., a_\ell)$ and witness $(a_{\ell+1}, ...a_m)$. The relation can be satisfied if there is a degree-bounded polynomial $h(X)$ such that:

$$\sum_{i=0}^{m} a_i u_i(X) \cdot \sum_{i=0}^{m} a_i v_i(X) = \sum_{i=0}^{m} a_i w_i(X) + h(X)t(X)$$

for prescribed polynomials $u_i(X), v_i(X), w_i(X), t(X)$. The SRS for the DV Groth16 consists of:

- αG, $\{x^i G\}_{i=1}^{n-1}$
- $r\beta G$, $\{rx^i G\}_{i=0}^{n-1}$
- $\{r^2(\beta u_i(x) + \alpha v_i(x) + w_i(x)) \cdot G\}_{i=\ell+1}^{m}$
- $\{r^2 x^i t(x) \cdot G\}_{i=0}^{n-2}$

The prover, knowing the public input and the witness, computes $P = P_1 + P_2 + P_3$ as follows:

$$P_1 = \alpha G + \sum_{i=0}^{m} a_i u_i(x) \cdot G$$
(using αG and $\{x^i G\}_{i=1}^{n-1}$)

$$P_2 = r\beta G + \sum_{i=0}^{m} a_i r v_i(x) \cdot G$$
(using $r\beta G$ and $\{rx^i G\}_{i=0}^{n-1}$)

$$P_3 = r^2 h(x)t(x) \cdot G + \sum_{i=\ell+1}^{m} a_i r^2(\beta u_i(x) + \alpha v_i(x) + w_i(x)) \cdot G$$
(using $\{r^2 x^i t(x) \cdot G\}_{i=0}^{n-2}$ and $\{r^2(\beta u_i(x) + \alpha v_i(x) + w_i(x)) \cdot G\}_{i=\ell+1}^{m}$)

Then, upon receiving P, the verifier responds with x, α, β to the prover, but does not reveal r. The prover then calculates and returns three scalars s_1, s_2, s_3 to the verifier.

$$s_1 = \alpha + \sum_{i=0}^{m} a_i u_i(x)$$
$$s_2 = \beta + \sum_{i=0}^{m} a_i v_i(x)$$
$$s_3 = h(x)t(x) + \sum_{i=\ell+1}^{m} a_i(\beta u_i(x) + \alpha v_i(x) + w_i(x))$$

The verifier checks that $P = (s_1 + r \cdot s_2 + r^2 \cdot s_3) \cdot G$ and that:

$$s_1 s_2 = \alpha\beta + s_3 + \sum_{i=0}^{\ell} a_i(\beta u_i(x) + \alpha v_i(x) + w_i(x))$$

DV 2-element Groth16. In the 2-element case, the relation is of a simpler form.

$$\left(\sum_{i=0}^{m} a_i u_i(X)\right)^2 = \sum_{i=0}^{m} a_i w_i(X) + h(X)t(X)$$

for prescribed polynomials $u_i(X), w_i(X), t(X)$. The SRS for the DV Groth16 consists of:

- αG and $\{x^i G\}_{i=1}^{n-1}$,
- $\{r(2\alpha u_i(x) + w_i(x)) \cdot G\}_{i=\ell+1}^{m}$
- $\{rx^i t(x) \cdot G\}_{i=0}^{n-2}$

The prover, knowing the public input and the witness, computes $P = P_1 + P_2$ as follows:

$$P_1 = \alpha G + \sum_{i=0}^{m} a_i u_i(x) \cdot G$$
(using αG and $\{x^i G\}_{i=1}^{n-1}$)

$$P_2 = rh(x)t(x) \cdot G + \sum_{i=\ell+1}^{m} a_i r(2\alpha u_i(x) + w_i(x)) \cdot G$$
(using $\{rx^i t(x) \cdot G\}_{i=0}^{n-2}$ and $\{r(2\alpha u_i(x) + w_i(x)) \cdot G\}_{i=\ell+1}^{m}$)

Then, upon receiving P, the verifier responds with x, α to the prover, but does not reveal r. The prover then calculates and returns two scalars s_1 and s_2 to the verifier.

$$s_1 = \alpha + \sum_{i=0}^{m} a_i u_i(x)$$
$$s_2 = h(x)t(x) + \sum_{i=\ell+1}^{m} a_i(2\alpha u_i(x) + w_i(x))$$

The verifier checks that $P = (s_1 + r \cdot s_2) \cdot G$ and that:

$$s_1^2 = \alpha^2 + s_2 + \sum_{i=0}^{\ell} a_i(2\alpha u_i(x) + w_i(x))$$

In summary, the BHIRW24 compiler consists of the following steps:

- **Prover ⇒ Verifier:** The prover commits the witness over the SRS. By carefully designing the SRS with a randomizer (r as above), one can generally aggregate the commitment into a single group element (P as above).
- **Verifier ⇒ Prover:** After receiving the commitment from the prover, the verifier reveals the secrets in SRS (which are x, α, β as above) except the randomizer r to the prover, so that the prover is able to compute the scalars and send them back to the verifier. The verifier checks that the commitment matches the scalars and checks the relations using the scalars.

Comparison with the BCIOP13 compiler. Compared to the other DV-SNARK compiler, the BCIOP13 compiler, this BHIRW24 compiler shares some similarities with the case using ElGamal encryption, but with one modification.

Previously, for ElGamal encryption, the difficulty is in decryption, which requires the computation of discrete log. To make the discrete log practical, a lot of restrictions have to be added.

The BHIRW24 compiler, as pointed out in [BHI+24], shifts the need to compute the discrete log to the prover, by revealing the necessary secrets in the SRS to the prover. The prover does not need to actually do the discrete log. The prover computes the scalars directly from the witness, thanks to the revealed secrets. The sacrifice is that the DV-SNARK setup becomes one-time.

The advantage that we do not need to worry about the practicality of discrete log removes a lot of tricky points around the DV-SNARK of ElGamal from the BCIOP13 compiler [BCI+13]. With

the BHIRW24 compiler, we do not need to restrict ourselves to smaller groups, we do not limit ourselves to inverse polynomial security (and have to resort to soundness amplification), and we do not need to require the statement to be verified in ZK to be expressed as a Boolean circuit.

Verify the SRS. The SRS can be verified in a way similar to that for the EPC SRS in § 5.1. We can first verify $\{x^iG\}_{i=1}^{n-1}$. Write $G_i = x^iG$. We can use the Chaum-Pedersen protocol, with respect to G, with a random number γ.

$$A = \sum_{i=1}^{n-2} \gamma^i \cdot G_i \qquad B = G_1 \qquad C = \sum_{i=1}^{n-2} \gamma^i \cdot G_{i+1}$$

We can similarly verify $\{rx^iG\}_{i=0}^{n-1}$. Write $H_i = rx^iG$. We can use the Chaum-Pedersen protocol, with respect to $H_0 = rG$, with the same random number.

$$A = \sum_{i=1}^{n-2} \gamma^i \cdot H_i \qquad B = H_1 \qquad C = \sum_{i=1}^{n-2} \gamma^i \cdot H_{i+1}$$

We check that $\{x^iG\}_{i=1}^{n-1}$ and $\{rx^iG\}_{i=0}^{n-1}$ are consistent by the Chaum-Pedersen protocol, with respect to G.

$$A = G_1 \qquad B = H_0 \qquad C = H_1$$

Given $\{x^iG\}_{i=1}^{n-1}$, it is possible to construct $u_i(x) \cdot G$, $v_i(x) \cdot G$, and $w_i(x) \cdot G$. Some random combinations and invocations of the Chaum-Pedersen protocol can verify consistency. In Groth16, without loss of generality, we can let $t(x) = x^n - 1$ for some n, and $t(x) \cdot G$ can be constructed and verified with the Chaum-Pedersen protocol. In summary, it is possible to prove that the SRS is constructed correctly without using pairings and with a constant number of invocations to the Chaum-Pedersen protocol.

Integration with Bitcoin script. To use DV-SNARK from the BHIRW24 compiler in BitVM, a few modifications to the protocol are needed.

First, we need to use the Bitcoin script to enforce the designated verifier (here, the challenger) to reveal the SRS secrets (for 3-element DV-Groth16, x, α, β, and for 2-element DV-Groth16, x, α) that are consistent with the SRS that is being verified. There are two necessary steps here.

- The challenger encrypts the SRS secrets under a random key K, computes its hash $h = H(K)$, and sends the ciphertext and hash h to the operator.
- The challenger creates a NIZK proof that shows that, given the ciphertext, the hash h, αG, xG, and βG for the 3-element DV-Groth16 as public input, there exists a key K such that $h = H(K)$ that can decrypt the ciphertext and reveal the underlying plaintext as x, α, β (for 3-element DV-Groth16) or x, α (for 2-element DV-Groth16). The operator verifies this NIZK proof.

Second, we need to reveal K to the operator, after the operator has submitted the commitment P. This is to enable the operator, who plays the role of the prover, to generate the scalars. This requires modifications to the original BitVM protocol.

- **Standard BitVM:** Note that we already need to borrow some techniques from reverse BitVM to enable multiple challengers, and we now just need to borrow one more technique from the reverse BitVM design that inserts an additional round after the **Assert-i, j** transaction. Now, in the Assert transaction, the prover commits the group element P as input labels to the GC. This is followed by another transaction from the challenger, which we can call **OpenSRS-i, j**. In such a transaction, the challenger is asked to reveal K. If the challenger did not reveal K before a prescribed period, this challenger cannot challenge this proof further. Then, the prover is asked to submit the scalars, in a new transaction which we can call **SubmitScalars-i, j**, also as input labels to the GC. With all the information, if the proof is invalid, the challenger can disprove by submitting another transaction.
- **Reverse BitVM:** The reverse BitVM would work in a similar way with the standard BitVM except that the GC is generated by the challengers and that the operator evaluates the GC with the correct proof to show that the proof is valid. Compared with the reverse BitVM for DV-SNARK with reusable preprocessing, here, for the BHIRW24 compiler, we need to add two more transaction. After the **SubmitSolution-i, j**, we need to ask the challenger to respond with K, which we can call **OpenSRS-i, j**, and then the operator needs to submit the scalars in a new transaction, which we can call **SubmitScalars-i, j**. The operator and the challenger continue with the original protocol to exchange the input labels.

Third, we have already mentioned that for DV-SNARK to work, we need to ensure that the input labels corresponding to the verifier secrets (x, α, β, r for 3-element DV-Groth16, and x, α, r for 2-element DV-Groth16) are correctly communicated, which requires that the NIZK from the challengers to show that the requested labels correspond to the SRS.

Implementation in GC. Implementing the circuit has some tricks, since it can use very large lookup tables to help verify the exponents (by scalar multiplication) which can be implemented efficiently with more tailored protocols [KP17; HK21] that one should not dismiss, in addition to known tricks on multiscalar multiplication (MSM) [BH23]. The implementation of Chainway Labs/Citrea and a few other teams [Cit25] suggested that a single scalar multiplication in the G1 group over the BN254 curve takes about 100 million gates (without optimization with lookup tables or the like [HK21]).

6 Operator liveness setup

Challengers help secure the BitVM protocol from malicious operators, but for the BitVM protocol to run correctly, we also need liveness, which neither challengers nor GC can help.

Importance of liveness. Lack of liveness can cause serious disruptions of the BitVM service. For example, say that a user Bob wants to withdraw the BTC locked in the BitVM protocol. If none of the operators are online (which could be intentional or unintentional), Bob may have to wait indefinitely. Specifically, operators may even censor Bob, so that they may willingly help other users withdraw their tokens, but not Bob. This could even be used to threaten Bob or demand a ransom from Bob, so that unless Bob bribes the operators, the operators will continue to refuse Bob. A less malicious operator may intentionally complicate or delay withdrawal to maintain a good-looking TVL. This is not uncommon in centralized blockchain applications.

This is why decentralization matters in the BitVM protocol, and we need multiple operators. We want to ensure that Bob can withdraw the BTC if some of the operators are online and cooperative, and not all the operators are required. This is also practical, as some operators that were included when Alice deposited BTC into the BitVM protocol may eventually become inactive when Bob plans to withdraw the locked BTC in the same BitVM instance.

It is useful to note that the liveness guarantee does not depend solely on the number of operators. The protocol seeks operators who can make decisions independently and strictly following protocols, rather than colluding with other operators. The protocol also seeks operators who implement mechanisms to guarantee their own liveness, such as having several replicates in different geographical regions, but having several replicates within an operator does not count as having many operators in the BitVM protocol.

Existential honesty. The standard BitVM setup that we commonly refer to uses *existential honesty*, meaning that as long as there is one honest operator, the protocol can progress and, specifically, Bob is guaranteed to be able to withdraw the BTC according to the protocol.

The existential honesty is a strength of BitVM compared with multi-signature in terms of liveness. Protocols based on multi-signature usually need an honest majority (half of the nodes) to work together. BitVM, however, only requires one honest operator to participate in the protocol. However, there is a cost. Due to the use of GC, the off-chain overhead increases linearly with the number of operators. This is why we want to share another operator liveness setup that can reduce the overhead, but it only works for reverse BitVM (therefore, with designated challengers and inherently designated verifiers).

Honest majority. If we consider a weaker operator liveness setup in which we require an honest majority of the operators to be available, we can reduce the off-chain cost to be independent from the number of operators (as if there is only one operator).

The idea is as follows. Let us start with reverse BitVM with a single operator. This operator has a secret key that can be used to sign transactions (see § 2.3 for transactions that require the signature of the corresponding operator), and whoever knows this key can also sign transactions on behalf of the operator.

Now, we replace this single operator with M nodes where each node owns a share of that secret key. Here, we can use multiparty ECDSA [CGG+20] that works with Shamir's secret sharing [Sha79], so that a majority of the nodes can use their shares to sign a transaction, without reconstructing or revealing the secret key.

These are not the only secrets the operator knows. In reverse BitVM, the operator knows the *reverse input labels*, which are used to exchange for actual input labels from the challengers using the **SubmitSolution-i, j** transaction and the **InputLabels-i, j** transaction. So, we need to modify how the reverse input labels are generated and how their hashes are computed.

- Using secure multiparty computation (MPC) [GMW87; GMW87], these M nodes can collectively generate the reverse input labels and compute their Shamir's secret shares and assign shares to nodes. In this process, a minority of the nodes cannot learn the reverse input labels. Here, we use Shamir's secret sharing to ensure that a majority of the nodes, rather than all M nodes, can use their shares to reconstruct the reverse input labels that are later revealed on-chain in the **SubmitSolution-i, j** transaction. It is sufficient to use MPC protocols that are maliciously secure, but assuming an honest majority.
- Next, using MPC, the nodes compute the hashes of each of the reverse input labels and reveal the hashes. These hashes would be shared with the challengers and be used to construct the transaction flows, as the hashes of reverse input labels (not the reverse input labels themselves) are hardcoded in these (presigned) transactions.

What happens next is that, when a majority of the nodes agree to submit the solution (they can check if the proof is generated correctly and whether the other co-signing nodes are following the protocols), they can use the Shamir's secret shares of the reverse input labels to reconstruct the reverse input labels, which are included in the **SubmitSolution-i, j** transaction.

The benefit of honest majority is that the off-chain cost for generating and evaluating the GC is as if there is only one operator. This is different from the existential honesty model in which several operators may be submitting their claims at the same time (only one of them could be honest), and therefore the number of copies of GC needs to grow with the number of the operators (and the number of challengers). Since there can only be one honest majority, only one claim can be made.

7 Discussion

In this section, we discuss some related ideas about garbling schemes and covenants.

7.1 Maliciously secure garbling

We now discuss two related concepts for achieving malicious security in garbled circuits and why they cannot immediately help with BitVM.

Authenticated garbling. We discussed how to use C&C to ensure that at least one of the circuits that the garbler generates is correct and usable. One may wonder why we are not using authenticated garbling [WRK17]. There are two reasons.

First, authenticated garbling schemes usually require interaction with evaluators when generating the circuits. For BitVM, it means that each circuit now has a designated challenger in that each challenger needs to have engaged with the garbler to run the authenticated garbling protocol, one by one, because they might not be able to trust another challenger's interaction with the garbler. This is not a deal breaker, as there are situations where there is only one challenger or a few challengers.

Second, probably a more fundamental limitation of authenticated garbling, even if we are okay with designated challengers, is that the evaluator cannot determine if the circuit is good before receiving the input labels.

- In [WRK17], the garbler (called $\mathsf{P_A}$ in the paper) can simply incorrectly generate the garbled table entries $G_{\gamma,0}$, $G_{\gamma,1}$, $G_{\gamma,2}$, $G_{\gamma,3}$ for a gate for the output wire γ. The evaluator cannot discover this inconsistency before evaluating this gate.
- In [LWYY25a], the garbler can simply claim an incorrect output label to be committed in the BitVM protocol. The evaluator can only find this out when it evaluates this output.

This does not violate the security of these schemes because they are designed for malicious security *with abort*. They ensure that if the garbler aborts the protocol, the garbler also does not learn any secret information from the evaluator, but the computation will fail to continue. If the abort is identifiable and it can emit a secret when the garbler is guilty, then we can use this secret and submit it on-chain to either reject the claim (in standard BitVM) or skip a challenger (in reverse BitVM). However, existing authenticated garbling protocols [WRK17; LWYY25a] are not known to have such a property.

C&C has the benefit that as long as the garbler passes the check, at least one of the remaining garbled circuits is expected to work. In contrast, security with abort does not protect against a garbler who intentionally wants to fail all garbled circuits.

Dual-execution garbling. Another idea to deal with a malicious garbler is *dual execution* [MF06]. This is a two-party computation protocol and would require a designated challenger. The idea is to have the operator generate a garbled circuit Γ_1 and have the challenger play the role of the garbler to generate another garbled circuit Γ_2 for the same computation. Assume that either the garbler or the challenger is honest and that one of Γ_1 and Γ_2 must be correct. Now, both parties reveal the input labels of the circuits that they garble, which allow both circuits to

be evaluated. If both circuits have the same output, then this is the correct output. If the output differs, then the computation fails to proceed and aborts.

There is a similar problem with this approach. The operator can simply generate a garbled circuit that is faulty and always outputs 1 (assuming that $C(x) = 0$ represents the case when the proof is invalid). This would prevent the challenger from being able to challenge the proof even when the proof is invalid, as the output would differ and the computation will abort.

7.2 Rerandomizable garbling

A relevant but indeed very different concept in garbling is called rerandomizable garbling [GHV10; HHJ25; AHKP22], which allows a rerandomizer to rerandomize an existing garbled circuit to a new one, and the rerandomized circuit cannot be distinguished from a newly generated circuit.

However, it appears that rerandomizable garbling does not help to reduce communication overhead in BitVM. If the garbler plays the role of rerandomizer, then the garbler still needs to send the rerandomized circuits. If the evaluator plays the role of a rerandomizer, then the evaluator knows the permutation to the keys and the labels and can effectively revert that permutation, which prevents one from reuse the garbled circuit by rerandomizing it again.

The issue we are facing is that rerandomization is still a private-key protocol. For rerandomizable garbling to be useful, we need a public-key protocol in which the evaluator can rerandomize the circuit without knowing any secret information, while the garbler has a secret key or a trapdoor that can generate the corresponding labels, which the evaluator would not be able to generate on its own.

7.3 Integration with covenants

If covenant opcodes (such as `OP_CAT`) are added to Bitcoin, NIZK might be able to integrate with covenants better than C&C and fraud proofs on Bitcoin. For BitVM to work correctly and securely, there are two verifications, both are necessary.

- **Check 1:** Before Alice deposits money, Alice needs to check that she is sending the money into a well-formed BitVM transaction sequence and confirming that all the operators and challengers have verified and received the GCs, and they are happy to work with this BitVM instance. Otherwise, the transaction sequence could be malicious and violate the BitVM protocol, or be ignored by operators and challengers.
- **Check 2:** When an operator submits a claim, at least one of the challengers needs to timely challenge the claim, carrying on the full challenge protocol, if the claim appears incorrect. Otherwise, a malicious operator might be able to claim Alice's deposit without paying it to Bob.

NIZK offers the following benefit: if the Bitcoin covenant is a state machine that, in each BitVM instance, moves from Step i to Step $(i + 1)$, it is possible to merge Check 1 of Step $(i + 1)$ with Check 2 of Step i, by making the following changes:

- The proof that is verified by the SNARK verifier not only checks the original spending condition (e.g. loan repayment on Ethereum), but also checks the following:

  - the new GC that will replace the GC for this ongoing SNARK verifier has been made available, for a sufficient period of time, on a prescribed data availability layer (such as Celestia [Cel] or EigenDA [Eigb]), so that the counterparties in the BitVM instance can download this new GC
  - the new GC is generated correctly and is guaranteed to work with the right labels, for which one can either use zero-knowledge proofs to verify all the gates in the GC are generated correctly, or assume a verifiable network like EigenCloud [Eiga].
  - The GCs are generated by the authorized parties (operators in standard BitVM and challengers in reverse BitVM).
- The covenant stores (by state carrying) the hashes of the input labels corresponding to the new GC that has been verified by the proof.

Therefore, if a user has validated that the covenant was correctly initialized and assuming that at least one challenger is available and honest, then the user does not need to check the covenant again in the future. The covenant can also delegate logic to the GC for computation, since the on-chain cost for the GC would be independent of the computation performed by the covenant.

7.4 Depegging prevention

If one uses BitVM to create a wrapped BTC (which can then be used to create other financial products out of it), it is important to prevent depegging. Deppeging occurs when many users want to withdraw their wrapped BTC at the same time, which could be due to many reasons. For example, users may simply want to convert the wrapped BTC back to BTC because they want to sell it for some liquidity. If there is a significant delay in this unwrapping, then users are less interested in holding the wrapped BTC anymore, and the price of the wrapped BTC may drop. To mitigate this issue, it is important to design the BitVM protocol with depegging prevention.

Who can withdraw. The protocol should be designed in a way that *any user* can unwrap a wrapped BTC into a regular BTC. This is more desirable than a restricted setting, as follows, where only the user who wraps that BTC can unwrap it back (like a *vault*). In such a restricted setting, when the wrapped BTC depegs, these users may not be incentivized to buy the wrapped BTC.

- As long as the user is not short on liquidity and does not have a need to sell that BTC within the challenge period, the user can always unwrap that wrapped BTC back at a later date to a BTC. Specifically, when the market lacks liquidity, BTC is also usually at a lower price, and it is not usually the best time to sell. Therefore, naturally, the user has no incentive to unwrap if nothing actually happens.
- A user who previously wrapped A BTC and currently holds B wrapped BTC (in one way or another) does not have much incentive to buy more than $(A - B)$ wrapped BTC from the market. Note that "in one way or another" may capture a lot of scenarios in DeFi. For example, a user may use the wrapped BTC as collateral to borrow more USD stablecoins that the user is expected to be able to repay (if the borrowed USD stablecoins are only used for low-risk DeFi products) or if the user has spent the stablecoins for personal expenses but is expected to pay back before the due. Note that a lending protocol is usually and sufficiently over-collateralized even if the underlying asset is the "digital gold" BTC, which matches a fairly bottom price of

BTC that is unlikely to reach, so we can expect that the user is incentivized to repay the loan and get the BTC back rather than wait for the loan to forfeit.

- As a result, users who may actually purchase wrapped BTC (at a below market price discount) are those users who have sold that wrapped BTC (or have permanently lost it, in one way or another, such as liquidation in perpetuals). For them, buying back the wrapped BTC is equivalent to buying a brand new BTC, and it would only be preferred when the wrapped BTC price is lower than the market BTC price, which usually happens only when the market is bearish and has limited liquidity. They can earn money if they have enough working capital and can *immediately* unwrap the coin back to the Bitcoin chain and then sell it, but be mindful that Bitcoin has a finality of about one hour, implying some risk that the token price may fluctuate in the meantime. It is useful to note that this *particular* user may just be reluctant to arbitrage (for example, not being online, or being distracted by other opportunities).

The situation can improve significantly if Bob can redeem a BTC that was initially deposited by another user, Alice. This would require the smart contract on another chain, such as Ethereum, to keep track of all BitVM instances and assign one for each withdrawal. There is some engineering overhead, but it should be manageable. This removes the restriction that only Alice can withdraw that specific wrapped BTC and Bob has no incentive to buy that wrapped BTC unless he has outstanding deposits that can be withdrawn. Now, as long as there are users in the network, which can be a market maker who is willing to arbitrage, one can expect the depegging to resolve.

How much to withdraw at once. Note that in the BitVM protocol, the operator has to help withdraw all the coins in a certain BitVM instance, and an operator cannot just front *part* of the capital. It is useful to set a cap on the maximum locked BTC of each BitVM instance so that none of them is insurmountably large (such as $1 billion USD worth of Bitcoin) to the extent that an operator finds it too difficult to front the capital.

Time to withdraw. An important advantage of BitVM is that as long as the operators are cooperative and have liquidity, users are expected to receive withdrawals almost immediately, since the operators front the capital. This is useful if a user (probably a market maker) wants to arbitrage, as it improves the liquidity.

That is to say, the challenge period does not directly impact the users, as the users would receive the BTC on the Bitcoin chain without the need to wait for the challenge period. However, the challenge period does impact the operators, and if all operators have their liquidity occupied, they would not be able to take any new withdrawal task, and a user Bob who wants to withdraw may have to wait. This could lead to a delay for end users (and encourage depegging).

To solve this, it is important to ensure that operators have liquidity and are *incentivized* to use liquidity for the BitVM protocol when users have a dire need to withdraw. One way to do so is to add a priority fee feature that is paid to the protocol and the operators during the period of time when there is a high demand for withdrawal, and another way is to auto-assign some tasks to capable operators if no operator volunteers to take, and have some suspension rules for operators who continuously fail to perform the tasks.

The fundamental solution to help operators with liquidity is to provide a way in which payments to operators can be made faster. This is possible in the designated verifier setting if all designated

verifiers are online and cooperative. We can design the protocol so that designated verifiers can explicitly express that they approve the claim on-chain, for example, by co-signing a special payout transaction, and if all the designated verifiers co-sign, the payout to the operators can be made immediate. The issue with this design is that it requires all designated verifiers to work together.

- A malicious designated verifier, who may, for example, financially benefit from depegging, can simply choose not to co-sign and keep the original challenge period.
- If the protocol wishes that all designated verifiers be online like this, it may limit how these verifiers are initially picked, which may limit the level of decentralization.
- When there are many designated verifiers, it is difficult to ensure that *all* designated verifiers are available. This is inherent to the "existential honesty" security for challengers.

If we cannot assume that all the designated verifiers would be online, then it is a matter of trade-off being (capital) efficiency and security to decide what the challenge period needs to be. It also helps to update the list of challengers in a timely manner to remove those that are not very active and add new ones, which can be done by letting the Ethereum smart contract try to prioritize withdrawing coins from the older BitVM instances or, more aggressively, to direct the operators to withdraw a coin straight into a new BitVM instance.

8 Applications

BitVM enables many applications that were previously impossible in Bitcoin. A notable example, suggested by Jeremy Rubin, is to apply BitVM to the zero-knowledge contingent payment (ZKCP) [Max11] and an extension called the zero-knowledge contingent service payment (ZKCSP) [CGGN17], now that BitVM also has succinct on-chain cost like ZKCP and ZKCSP.

8.1 Improved ZKCP

We first present the necessary background on ZKCP and discuss the problem of "buyer's cold feet" that could arise in certain situations (but not all) of ZKCP.

ZKCP. Consider a seller who knows a secret solution x so that $f(x) = 1$ and a buyer who wants to purchase this solution. They want a fair exchange with the following guarantees.

- if the seller receives the money from the buyer, the buyer is guaranteed to know x, such that $f(x) = 1$.
- if the buyer knows x without learning it from somewhere else, the seller is guaranteed to receive the money.

Such a fair exchange used to be difficult to realize without a trusted third party, but Bitcoin can replace this trusted third party, by using a hash-locked transaction and a zero-knowledge proof (which does not need to be an NIZK), as follows.

- The seller samples a symmetric encryption key K, creates a ciphertext of x as $\mathsf{ct}_x = E(K; x)$, hashes the key as $h = H(K)$, and creates a zero-knowledge proof showing that, given public input (ct_x, h), the seller knows a key K such that $h = H(K)$, $\mathsf{ct}_x = E(K; x)$, and $f(x) = 1$, without revealing K or x.
- The buyer verifies the zero-knowledge proof, and if the buyer decides to proceed with the purchase, it sends the payment to a UTXO with a Bitcoin script, which can be spent by the seller upon the seller revealing K' such that $H(K') = h$ (implying that $K' = K$) and signing on that transaction. As K is revealed, the seller can decrypt ct_x and obtain the solution x. The Bitcoin script also has a refund path: If the seller did not spend that UTXO within a period of time, the buyer can spend the UTXO and claim the money back.

The protocol provides both guarantees defined above.

Buyer's cold feet. Now we consider the situations in which the seller must put a lot of effort into finding x that satisfies $f(x) = 1$. This can be the case if the buyer is asking the seller to, for example, solve a Bitcoin Proof of Work (PoW). The buyer and the seller can still use the ZKCP protocol, but now there is a caveat.

For the ZKCP protocol to start, the seller must first solve for the solution x, and then the seller can encrypt this solution and generate a zero-knowledge proof showing that $f(x) = 1$.

Now, consider a situation where the buyer asked the seller to solve the PoW, the seller did solve the PoW before the promised deadline, but the buyer now changes mind, does not want to buy anymore, and is unwilling to respond to the seller in the ZKCP protocol. In this situation, the

seller does not receive the payment, and the buyer does not receive the solution to Bitcoin PoW. However, note that the seller has spent significant effort trying to solve the PoW, the seller now is operating at a loss.

ZKCP without buyer's cold feet. Therefore, we want a new protocol in which the seller has a guarantee, before finding the solution, that if the solution can be found, the buyer is guaranteed to purchase this solution upon presenting this solution.

This would also be useful for using ZKCP for *bug bounty* (suggested in [CGGN17]) where the white-hat hacker spends time hunting the bug and claims the bounty on-chain. It is possible that the buyer decided not to pay the seller for information about the bug but instead to find the bug themselves, with a free hint from the white-hat hacker that a bug exists.

More specifically, let us define this new kind of ZKCP as follows. Consider a buyer who is interested in knowing a solution x such that $f(x) = 1$ for a problem f and a seller who can provide this solution. They want a fair exchange in which the seller can ensure that, if the solution is found, the buyer is guaranteed to pay. This can be implemented with BitVM with a purpose-built Challenge-Assert mechanism as follows.

- Assume that the buyer (as the garbler) and the seller (as the evaluator) have correctly set up a garbled circuit Γ for $C(\pi, x)$ that takes a SNARK proof π on $f(x) = 1$ and the value x as input, and $C(\pi, x)$ outputs 0 if and only if the proof is valid given x as public input. Note that the use of SNARK is optional, as it might sometimes be easier to compute $f(x)$ directly, in which case we simply say that $\pi = \epsilon$.
- Say that the size of the input for $C(\pi, x)$ is ℓ. The seller generates ℓ pairs of seller's labels $\{S_i^{(0)}, S_i^{(1)}\}_{i\in[\ell]}$ and sends their hashes $s_i^{(0)} = H(S_i^{(0)})$ and $s_i^{(1)} = H(S_i^{(1)})$ to the buyer.
- The seller and the buyer co-sign the following chain of transactions (forming a covenant).
  - **SubmitSolution**: the seller can invoke the SubmitSolution transaction by opening one of the two hashes $s_i^{(0)}$ and $s_i^{(1)}$ for every $i \in [\ell]$. The revealed bits correspond to the seller's proof π and the solution x, which are inputs to $C(\pi, x)$, but note that $S_i^{(b_i)}$ are not the input labels to the garbled circuit Γ.
  - **InputLabels**: upon receiving the SubmitSolution transaction, the buyer is required to open $h_i^{(b_i)}$ corresponding to the opened hash $s_i^{(b_i)}$. This can be implemented by asking the buyer to reveal $(s_i^{(b_i)}, h_i^{(b_i)})$ in pairs, and since the buyer only knows $S_i^{(b_i)}$ revealed in the SubmitSolution transaction, it has to reveal the corresponding $L_i^{(b_i)}$. If the buyer fails to respond within a prescribed time period, it triggers time-out, and the seller can invoke a transaction to claim the payout. Note that if the buyer is satisfied with the answer, the buyer does not have to respond and can safely time out, in which case the seller still receives the money, and the circuit Γ can be used for the next time.
  - **NormalPayout**: upon receiving the InputLabels transaction, the seller can run Γ with the input labels $\{L_i^{(b_i)}\}_{i\in[\ell]}$ to obtain the output label for value 0, which enables the seller to get the payout. Note that if the seller did not submit a correct solution with the proof, it would not obtain the output label for value 0, and it would not receive the payout.
- If the seller solves the problem, it submits the solution through the SubmitSolution transaction.

- If the buyer is happy with the solution, the buyer can simply not respond and wait for it to time out, so that the seller can get the payout.
- Otherwise, if the buyer is unhappy with the solution, the buyer responds with the InputLabels transaction before the prescribed timeout. In this situation, if the seller admits that the solution is invalid, the seller can simply choose not to evaluate Γ since it is futile. However, if the seller believes that the solution is correct, the seller can run Γ and claim the payout using the output label for 0.

On-chain privacy. If the seller and buyer do not want to reveal the solution x in plaintext on the blockchain, they can agree on a random one-time pad $\tau \leftarrow \{0,1\}^\ell$, and instead of listing $s_i^{(0)}$ before $s_i^{(1)}$ and $h_i^{(0)}$ before $h_i^{(1)}$ in the Bitcoin script, the order is changed according to τ, so $s_i^{(\tau_i)}$ appears before $s_i^{(1-\tau_i)}$ and $h_i^{(\tau_i)}$ appears before $h_i^{(1-\tau_i)}$. This is in essence similar to "color bits" in garbled circuits.

Dealing with a long input. A limitation of this approach is that the size of the input ℓ to Γ increases with the bit length of the solution x. It is possible to implement a compression hash function in the Bitcoin script to reduce the length of the input to Γ, but it could increase the on-chain cost.

Finally, to summarize, one can combine this protocol with the standard ZKCP, so that if the buyer does not have cold feet, the buyer and the seller go ahead with the standard ZKCP protocol in which the seller is also asked to disclose some sort of a "cancellation key" that can be used to cancel the no-cold-feet protocol above so that the buyer can use it to avoid double payments in the no-cold-feet protocol. This is much easier to execute than the no-cold-feet protocol.

8.2 Improved ZKCSP

The construction for ZKCP described above naturally applies to ZKCSP, which extends ZKCP to certain applications for which the original ZKCP does not work.

In ZKCSP, the buyer is not thinking about finding a solution to a problem, but about whether or not a certain service has been performed. We restrict the services here to those that can be verified with zero-knowledge proofs. The seller is expected to claim the payout only if the seller proves that the service has been performed.

The original ZKCP protocol (with the issue of buyer's cold feet) does not work well in this setting. In that protocol, the seller needs to first generate the proof (say Proof 1), encrypt the proof, and generate another proof (say Proof 2) that the proof in the ciphertext is a valid proof. The buyer can then decide whether or not to buy this proof and, if so, the buyer can create the hash-locked transaction in which the seller can only consume by revealing the encrypted key.

However, there is a caveat here—the buyer does not actually need to pay. The fact that the seller can generate a valid Proof 2 means that Proof 1 is also valid, and the service has already been performed. The buyer does not need to engage with the rest of the protocol to obtain Proof 1 to verify, since Proof 2 also implies that Proof 1 must be valid.

The ZKCSP protocol. ZKCSP [CGGN17] shows a way to remedy the issue, but not a complete fix. The idea of ZKCSP is to make Proof 2 an A-OR-B proof. Previously, Proof 2 shows that the

following three things are true simultaneously: $h = H(K)$, $\mathsf{ct}_x = E(K; x)$, and $f(x) = 1$. We do not really need ct_x here, so we drop it. Now, the A-OR-B proof requires that either A is true or B is true, as follows.

- A: $h = H(K)$ and Proof 1 is valid i.e. $f(x) = 1$
- or B: $h = H'(K)$

where H' is another hash function such that H and H' are claw-free, in that it is hard to find x and y such that $H(x) = H'(y)$. We also require that the distributions of the output of H and H' over random inputs are computationally indistinguishable. This can be instantiated by hash functions modeled as random oracles with domain separation. The zero-knowledge proof guarantees that a valid proof does not leak if A or B is true. In this case, Proof 2 itself does not reveal whether Proof 1 is valid. The buyer needs to make a hash-locked UTXO that can be spent only if the seller presents $K' = K$ such that $h = H(K')$.

If the seller can spend the UTXO (by revealing $K' = K$), then the buyer can confirm that it is case A; otherwise, the buyer does not learn anything, and the seller does not get the payment.

There are still three issues with this protocol, discussed as follows, that can affect certain use cases.

Buyer's luck. The first issue is related to the buyer's luck, in which the buyer does not pay but simply feels lucky that the seller actually performed the service. This may actually happen. If the seller is honest, then the seller would have actually performed the service. Of course, the buyer cannot be 100% sure that the service has been performed, but the buyer is not paying anyway.

A honest seller can try to mitigate this risk by making more false claims to test if the buyer actually wants to pay before actually performing the service. However, this requires that the same protocol is repeated many times. Since the protocol requires interaction between both parties on the Bitcoin network and the Bitcoin network has a long block time, it is not ideal. Moreover, there is still the possibility that the seller performed the service but did not receive the payment.

Buyer's side knowledge. The second issue that is not favorable to the seller is when the service performed by the seller is visible from outside. For example, if the service is to ask the seller to send a transaction on a blockchain, the buyer may already be able to observe that from the blockchain and does not need a proof from the seller to confirm, in which case the buyer can simply not pay.

This can be mitigated only if the seller can make the service itself in a "hash-locked" fashion, where the service can only be activated by the buyer using the key K in ZKCSP (or simply ZKCP) before a prescribed deadline, and if the buyer did not activate the service by the deadline, the seller can invoke a refund path that reverses the service in a way that does no harm to the seller. This requires the service to be deployed in a way that supports such a "hash-locked" feature, such as on an EVM chain, but the assumption is very restrictive. Note that if the service is deployed in a hash-locked format, one can simply use the standard ZKCP protocol.

It is also tricky whether the service can be reversed in a way that the seller does not suffer a loss. For example, if the seller plans to swap some tokens on ETH with the buyer's BTC, but the buyer decides not to proceed with the deal, the seller can still get the tokens back, but the fact that the tokens were locked for a period of time means that the seller may have lost opportunities

to earn yield or sell the tokens. This can be a problem for the seller if the token price fluctuates significantly in the meantime.

Buyer's cold feet. The third issue is the same as before, in which the buyer no longer cares about the service and simply wants to cancel it, but the seller could have already done the service, and even if the seller immediately reverses the service, the seller could still have a loss.

Solution using the no-cold-feet ZKCP protocol. The problem we encounter in the ZKCSP protocol can be solved using the no-cold-foot protocol described above, as a special case where $x = \epsilon$, meaning that the buyer does not need to know x. This provides a stronger protection to the seller because the seller can first ensure that as soon as it performs the service, the seller can receive the money from the buyer.

Acknowledgment

This paper is a literature review, as it recites the results from prior work, and credits should go to the large number of cryptographers who have contributed to the study of these primitives in the past two decades and researchers who have contributed to the study of BitVM with succinct on-chain cost.

The author thanks Ekrem Bal from Citrea for discussion on cut and choose, thanks Jeremy Rubin from Char Network for discussion related to LEGO-ing the garbled circuits and zero-knowledge contingent payment (ZKCP), and thanks David Tse, Srivatsan Sridhar, and Pingzhou Yuan from Babylon Labs for discussion related to NIZK for GC.

The author thanks Xiao Wang from Northwestern University for answering some questions about BitGC, thanks Hang Su, one of the authors of [ISW21], for discussion on multi-theorem soundness and parameterization in the case when zero knowledge is not required, and thanks Alireza Shirzad, one of the authors of Pari [DMS24], for discussion on alternative verifier implementation, which applies to Polymath, Pari, and Eagen25.

The author thanks cyphersnake from Babylon Labs regarding the performance numbers of the Groth16 verifier circuit.

The author also thanks Stephen Duan from GOAT network and Sanjam Garg from UC Berkeley for discussion and feedback.

References

[AAL+24] L. Aumayr, Z. Avarikioti, R. Linus, M. Maffei, A. Pelosi, C. Stefo, and A. Zamyatin, "BitVM: Quasi-Turing complete computation on Bitcoin," in *ePrint 2024/1995*. [Online]. Available: `https://eprint.iacr.org/2024/1995.pdf`.

[ADI25] G. Arnon, J. Dujmovic, and Y. Ishai, "Designated-verifier SNARGs with one group element," in *CRYPTO '25*. [Online]. Available: `https://eprint.iacr.org/2025/517.pdf`.

[AHKP22] A. Acharya, C. Hazay, V. Kolesnikov, and M. Prabhakaran, "SCALES: MPC with small clients and larger ephemeral servers," in *TCC '22*. [Online]. Available: https://eprint.iacr.org/2022/751.pdf.

[AMPR14] A. Afshar, P. Mohassel, B. Pinkas, and B. Riva, "Non-interactive secure computation based on cut-and-choose," in *EUROCRYPT '14*. [Online]. Available: https://eprint.iacr.org/2015/282.pdf.

[AMPŠ24] T. Ashur, M. Mahzoun, J. Posen, and D. Šijačić, "Vision Mark-32: ZK-friendly hash function over binary tower fields," in *ePrint 2024/633*. [Online]. Available: https://eprint.iacr.org/2024/633.pdf.

[BA25] O. Biçer and A. Ajorian, "AuthOr: Lower cost authenticity-oriented garbling for arbitrary boolean circuits," in *ePrint 2025/775*. [Online]. Available: https://eprint.iacr.org/2025/775.pdf.

[BCCT13] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "Recursive composition and bootstrapping for SNARKs and proof-carrying data," in *STOC '13*. [Online]. Available: https://eprint.iacr.org/2012/095.pdf.

[BCF+25] M. Brehm, B. Chen, B. Fisch, N. Resch, R. D. Rothblum, and H. Zeilberger, "Blaze: Fast SNARKs from interleaved RAA codes," in *EUROCRYPT '25*. [Online]. Available: https://eprint.iacr.org/2024/1609.pdf.

[BCG+17] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, and M. Orrù, "Homomorphic secret sharing: Optimizations and applications," in *CCS '17*. [Online]. Available: https://eprint.iacr.org/2018/419.pdf.

[BCG+22] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, N. Resch, and P. Scholl, "Correlated pseudorandomness from expand-accumulate codes," in *CRYPTO '22*. [Online]. Available: https://eprint.iacr.org/2022/1014.pdf.

[BCI+13] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth, "Succinct non-interactive arguments via linear interactive proofs," in *TCC '13*. [Online]. Available: https://eprint.iacr.org/2012/718.pdf.

[BCO+21] A. Ben-Efraim, K. Cong, E. Omri, E. Orsini, N. P. Smart, and E. Soria-Vazquez, "Large scale, actively secure computation from LPN and free-XOR garbled circuits," in *EUROCRYPT '21*. [Online]. Available: https://eprint.iacr.org/2021/120.pdf.

[BD19] R. Barbulescu and S. Duquesne, "Updating key size estimations for pairings," in *JoC '19*.

[BEK+20] D. Božilov, M. Eichlseder, M. Kneževic, B. Lambin, G. Leander, T. Moos, V. Nikov, S. Rasoolzadeh, Y. Todo, and F. Wiemer, "PRINCEv2: More security for (almost) no overhead," in *SAC '20*. [Online]. Available: https://eprint.iacr.org/2020/1269.pdf.

[BFK+24] A. R. Block, Z. Fang, J. Katz, J. Thaler, H. Waldner, and Y. Zhang, "Field-agnostic SNARKs from expand-accumulate codes," in *CRYTPTO '24*. [Online]. Available: https://eprint.iacr.org/2024/1871.pdf.

[BGG+14] D. Boneh, C. Gentry, S. Gorbunov, S. Halevi, V. Nikolaenkok, G. Segev, V. Vaikuntanathan, and D. Vinayagamurthy, "Fully key-homomorphic encryption, arithmetic circuit ABE, and compact garbled circuits," in *EUROCRYPT '14*. [Online]. Available: https://eprint.iacr.org/2014/356.pdf.

[BGH+23] G. Beck, A. Goel, A. Hegde, A. Jain, Z. Jin, and G. Kaptchuk, "Scalable multiparty garbling," in *CCS '23*. [Online]. Available: https://eprint.iacr.org/2023/099.pdf.

[BGI16] E. Boyle, N. Gilboa, and Y. Ishai, "Breaking the circuit size barrier for secure computation under DDH," in *CRYPTO '16*. [Online]. Available: https://eprint.iacr.org/2016/585.pdf.

[BGI17] E. Boyle, N. Gilboa, and Y. Ishai, "Group-based secure computation: Optimizing rounds, communication, and computation," in *EUROCRYPT '17*. [Online]. Available: https://eprint.iacr.org/2017/150.pdf.

[BGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation," in *STOC '88*. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/62212.62213.

[BH23] G. Botrel and Y. E. Housni, "EdMSM: Multi-scalar-multiplication for SNARKs and faster Montgomery multiplication," in *CHES '23*. [Online]. Available: https://eprint.iacr.org/2022/1400.pdf.

[BHI+24] N. Bitansky, P. Harsha, Y. Ishai, R. D. Rothblum, and D. J. Wu, "Dot-product proofs and their applications," in *FOCS '24*. [Online]. Available: https://eprint.iacr.org/2024/1138.pdf.

[BIOW20] O. Barta, Y. Ishai, R. Ostrovsky, and D. J. Wu, "On succinct arguments and witness encryption from groups," in *CRYPTO '20*. [Online]. Available: https://eprint.iacr.org/2020/1319.pdf.

[BL25] D. J. Bernstein and T. Lange, "Safe curves for elliptic-curve cryptography," in Springer, ch. Information security in a connected world: Celebrating the life and work of Ed Dawson.

[BM93] J. Benaloh and M. de Mare, "One-way accumulators: A decentralized alternative to digital signatures," in *EUROCRYPT '93*. [Online]. Available: https://link.springer.com/content/pdf/10.1007/3-540-48285-7_24.pdf.

[BSCG+13] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *CRYPTO '13*. [Online]. Available: https://eprint.iacr.org/2013/507.pdf.

[BSS+15] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK lightweight block ciphers," in *DAC '15*. [Online]. Available: https://eprint.iacr.org/2013/404.pdf.

[BTVW17] Z. Brakerski, R. Tsabary, V. Vaikuntanathan, and H. Wee, "Private constrained PRFs (and more) from LWE," in *TCC '17*. [Online]. Available: https://eprint.iacr.org/2017/795.pdf.

[BV11] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," in *FOCS '11*. [Online]. Available: https://eprint.iacr.org/2011/344.

[BZ24] K. Brazitikos and V. Zikas, "General adversary structures in Byzantine agreement and multi-party computation with active and omission corruption," in *TCC '24*. [Online]. Available: https://eprint.iacr.org/2024/209.pdf.

[Ben84] J. Benaloh, "Dense probabilistic encryption," in *SAC '94*. [Online]. Available: https://sacworkshop.org/proc/SAC_94_006.pdf.

[Bra13] L. T. A. N. Brandão, "Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique," in *ASIACRYPT '13*. [Online]. Available: https://eprint.iacr.org/2013/577.pdf.

[CGG+20] R. Canetti, R. Gennaro, S. Goldfeder, N. Makriyannisk, and U. Peled, "UC non-interactive, proactive, distributed ECDSA with identifiable aborts," in *CCS '20*. [Online]. Available: https://eprint.iacr.org/2021/060.pdf.

[CGGN17] M. Campanelli, R. Gennaro, S. Goldfeder, and L. Nizzardo, "Zero-knowledge contingent payments revisited: Attacks and payments for services," in *CCS '17*. [Online]. Available: https://eprint.iacr.org/2017/566.pdf.

[CHM+20] A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. Ward, "Marlin: Preprocessing zkSNARKs with universal and updatable SRS," in *EUROCRYPT '20*. [Online]. Available: https://eprint.iacr.org/2019/1047.pdf.

[CO15] T. Chou and C. Orlandi, "The simplest protocol for oblivious transfer," in *LATINCRYPT '15*. [Online]. Available: https://eprint.iacr.org/2015/267.pdf.

[CP92] D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *CRYPTO '92*. [Online]. Available: https://link.springer.com/content/pdf/10.1007/3-540-48071-4_7.pdf.

[Cel] *Celestia: The modular blockchain network*, https://celestia.org/.

[Cit25] Citrea, *Garbled SNARK verifier circuit*, https://github.com/BitVM/garbled-snark-verifier.

[DBP96] H. Dobbertin, A. Bosselaers, and B. Preneel, "RIPEMD-160: A strengthened version of RIPEMD," in *FSE '96*. [Online]. Available: https://link.springer.com/content/pdf/10.1007/3-540-60865-6_44.pdf.

[DJ01] I. Damgärd and M. Jurik, "A generalisation, a simplification and some applications of Paillier's probabilistic public-key system," in *PKC '01*. [Online]. Available: https://link.springer.com/content/pdf/10.1007/3-540-44586-2_9.pdf.

[DJM98] D. Divsalar, H. Jin, and R. J. McEliece, "Coding theorems for "turbo-like" codes," in *Allerton Conference on Communication, Control, and Computing '98*.

[DL78] R. A. Demillo and R. J. Lipton, "A probabilistic remark on algebraic program testing," in *Information Processing Letters '78*.

[DMS24] M. Dellepere, P. Mishra, and A. Shirzad, "Garuda and pari: Faster and smaller SNARKs via equifficient polynomial commitments," in *ePrint 2024/1245*. [Online]. Available: https://eprint.iacr.org/2024/1245.pdf.

[DP23] B. E. Diamond and J. Posen, "Proximity testing with logarithmic randomness," in *ePrint 2023/630*. [Online]. Available: https://eprint.iacr.org/2023/630.pdf.

[DP24] B. E. Diamond and J. Posen, "Polylogarithmic proofs for multilinears over binary towers," in *ePrint 2024/504*. [Online]. Available: https://eprint.iacr.org/2024/504.pdf.

[DP25] B. E. Diamond and J. Posen, "Succinct arguments over towers of binary fields," in *EUROCRYPT '25*. [Online]. Available: https://eprint.iacr.org/2023/1784.pdf.

[DR02] J. Daemen and V. Rijmen, *The Design of Rijndael AES: The Advanced Encryption Standard*. Springer-Verlag.

[DT08] I. Damgard and N. Triandopoulos, "Supporting non-membership proofs with bilinear-map accumulators," in *ePrint 2008/538*. [Online]. Available: https://eprint.iacr.org/2008/538.pdf.

[Eag25] L. Eagen, "Glock: Garbled locks for Bitcoin," in *ePrint 2025/1485*. [Online]. Available: https://eprint.iacr.org/2025/1485.

[Eiga] *Eigencloud*, https://www.eigencloud.xyz/.

[Eigb] *EigenDA*, https://www.eigenda.xyz/.

[ElG84] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *CRYPTO '84*. [Online]. Available: https://link.springer.com/content/pdf/10.1007/3-540-39568-7_2.pdf.

[FJNT16] T. K. Frederiksen, T. P. Jakobsen, J. B. Nielsen, and R. Trifiletti, "On the complexity of additively homomorphic UC commitments," in *TCC '16*. [Online]. Available: https://eprint.iacr.org/2015/694.pdf.

[FNO15] T. K. Frederiksen, J. B. Nielsen, and C. Orlandi, "Privacy-free garbled circuits with applications to efficient zero-knowledge," in *EUROCRYPT '15*. [Online]. Available: https://eprint.iacr.org/2014/598.pdf.

[Fia24] Fiamma, *Zk proof verification on Bitcoin with BitVM2 for the price of a coffee*. [Online]. Available: https://x.com/fiamma_labs/status/1828499846153507180.

[GGM84] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," in *FOCS '84*. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/6490.6503.

[GGPR13] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct NIZKs without PCPs," in *EUROCRYPT '13*. [Online]. Available: https://eprint.iacr.org/2012/215.pdf.

[GHV10] C. Gentry, S. Halevi, and V. Vaikuntanathan, "i-Hop homomorphic encryption and rerandomizable Yao circuits," in *CRYPTO '10*. [Online]. Available: https://eprint.iacr.org/2010/145.pdf.

[GKL+25] L. Grassi, D. Khovratovich, R. Lüftenegger, C. Rechberger, M. Schofnegger, and R. Walch, "Monolith: Circuit-friendly hash functions with new nonlinear layers for fast and constant-time implementations," in *ToSC/FSE '25*. [Online]. Available: https://eprint.iacr.org/2023/1025.pdf.

[GKM+21] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. S. Thomsen, *Grøstl: A SHA-3 candidate*, https://www.groestl.info/Groestl.pdf.

[GKP+13] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich, "Reusable garbled circuits and succinct functional encryption," in *STOC '13*. [Online]. Available: https://eprint.iacr.org/2012/733.pdf.

[GKR08] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: Interactive proofs for muggles," in *STOC '08*. [Online]. Available: https://eccc.weizmann.ac.il/report/2017/108/.

[GKS23] L. Grassi, D. Khovratovich, and M. Schofnegger, "Poseidon2: A faster version of the Poseidon hash function," in *AFRICACRYPT '23*. [Online]. Available: https://eprint.iacr.org/2023/323.pdf.

[GKW+20] C. Guo, J. Katz, X. Wang, C. Weng, and Y. Yu, "Better concrete security for half-gates garbling (in the multi-instance setting)," in *CRYPTO '20*. [Online]. Available: https://eprint.iacr.org/2019/1168.pdf.

[GKWY20] C. Guo, J. Katz, X. Wang, and Y. Yu, "Efficient and secure multiparty computation from fixed-key block ciphers," in *S&P '20*. [Online]. Available: https://eprint.iacr.org/2019/074.pdf.

[GLS+23] A. Golovnev, J. Lee, S. Setty, J. Thaler, and R. S. Wahby, "Brakedown: Linear-time and field-agnostic SNARKs for R1CS," in *CRYPTO '23*. [Online]. Available: https://eprint.iacr.org/2021/1043.pdf.

[GM08] V. Guruswami and W. Machmouchi, "Explicit interleavers for a repeat accumulate accumulate (RAA) code construction," in *ISIT '08*. [Online]. Available: https://doi.org/10.1109/ISIT.2008.4595333.

[GMNO18] R. Gennaro, M. Minelli, A. Nitulescu, and M. Orrù, "Lattice-based zk-SNARKs from square span programs," in *CCS '18*. [Online]. Available: https://eprint.iacr.org/2018/275.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson, "How to play ANY mental game, or a completeness theorem for protocols with honest majority," in *STOC '87*. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/28395.28420.

[GSW13] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *CRYPTO '13*. [Online]. Available: https://eprint.iacr.org/2013/340.pdf.

[GW21] A. Gabizon and Z. J. Williamson, "fflonk: A fast-Fourier inspired verifier efficient version of PlonK," in *ePrint 2021/1167*. [Online]. Available: https://eprint.iacr.org/2021/1167.pdf.

[GWC19] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge," in *ePrint 2019/953*. [Online]. Available: https://eprint.iacr.org/2019/953.pdf.

[Gro16] J. Groth, "On the size of pairing-based non-interactive arguments," in *EUROCRYPT '16*. [Online]. Available: https://eprint.iacr.org/2016/260.pdf.

[HHJ25] R. Heitjohann, J. von der Heyden, and T. Jager, "Rerandomizable garbling, revisited," in *CRYPTO '25*. [Online]. Available: https://eprint.iacr.org/2025/843.pdf.

[HK21] D. Heath and V. Kolesnikov, "One hot garbling," in *CCS '21*. [Online]. Available: https://eprint.iacr.org/2022/798.pdf.

[HKE13] Y. Huang, J. Katz, and D. Evans, "Efficient secure two-party computation using symmetric cut-and-choose," in *CRYPTO '13*. [Online]. Available: https://eprint.iacr.org/2013/081.pdf.

[HKK+14] Y. Huang, J. Katz, V. Kolesnikov, R. Kumaresan, and A. J. Malozemoff, "Amortizing garbled circuits," in *CRYPTO '14*. [Online]. Available: https://eprint.iacr.org/2015/081.pdf.

[HLL23] Y.-C. Hsieh, H. Lin, and J. Luo, "Attribute-based encryption for circuits of unbounded depth from lattices: Garbled circuits of optimal size, laconic functional evaluation, and more," in *FOCS '23*. [Online]. Available: https://eprint.iacr.org/2023/1716.pdf.

[HLP24] U. Haböck, D. Levit, and S. Papini, "Circle STARKs," in *ePrint '24*. [Online]. Available: https://eprint.iacr.org/2024/278.pdf.

[HS24] T. den Hollander and D. Slamanig, "A crack in the firmament: Restoring soundness of the Orion proof system and more," in *ePrint 2024/1164*. [Online]. Available: https://eprint.iacr.org/2024/1164.pdf.

[Hab22] U. Haböck, "Multivariate lookups based on logarithmic derivatives," in *ePrint 2022/1530*.

[Hat] C. Hatch, *Hashed-timelock-contract-ethereum*, https://github.com/chatch/hashed-timelock-contract-ethereum.

[ILL25a] Y. Ishai, H. Li, and H. Lin, "A unified framework for succinct garbling from homomorphic secret sharing," in *CRYPTO '25*. [Online]. Available: https://eprint.iacr.org/2025/442.pdf.

[ILL25b] Y. Ishai, H. Li, and H. Lin, "Succinct homomorphic MACs from groups and applications," in *FOCS '25*. [Online]. Available: https://eprint.iacr.org/2024/2073.pdf.

[ISW21] Y. Ishai, H. Su, and D. J. Wu, "Shorter and faster post-quantum designated-verifier zkSNARKs from lattices," in *CCS '21*. [Online]. Available: https://eprint.iacr.org/2021/977.pdf.

[Imp95] R. Impagliazzo, "A personal view of average-case complexity," in *CCC '95*.

[JKW+23] P. Jain, S. Kumar, S. Wooders, S. G. Patil, J. E. Gonzalez, and I. Stoica, "Skyplane: Optimizing transfer cost and throughput using cloud-aware overlays," in *NSDI '23*. [Online]. Available: https://www.usenix.org/system/files/nsdi23-jain.pdf.

[KL24] V. Y. Kemmoe and A. Lysyanskaya, "RSA-based dynamic accumulator without hashing into primes," in *CCS '24*. [Online]. Available: https://eprint.iacr.org/2024/505.pdf.

[KM11] J. Katz and L. Malka, "Constant-round private function evaluation with linear complexity," in *ASIACRYPT '11*. [Online]. Available: https://eprint.iacr.org/2010/528.

[KMN23] G. Kadianakis, M. Maller, and A. Novakovic, "Sigmabus: Binding sigmas in circuits for fast curve operations," in *ePrint 2023/1406*. [Online]. Available: https://eprint.iacr.org/2023/1406.

[KNR+17] V. Kolesnikov, J. B. Nielsen, M. Rosulek, N. Trieu, and R. Trifiletti, "DUPLO: Unifying cut-and-choose for garbled circuits," in *CCS '17*. [Online]. Available: https://eprint.iacr.org/2017/344.pdf.

[KP17] Y. Kondi and A. Patra, "Privacy-free garbled circuits for formulas: Size zero and information-theoretic," in *CRYPTO '17*. [Online]. Available: https://eprint.iacr.org/2017/561.pdf.

[KS08] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *ICALP '08*. [Online]. Available: https://www.cs.toronto.edu/~vlad/papers/XOR_ICALP08.pdf.

[KZG10] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *ASIACRYPT '10*. [Online]. Available: https://www.iacr.org/archive/asiacrypt2010/6477178/6477178.pdf.

[Kob91] N. Koblitz, "CM-curves with good cryptographic properties," in *CRYPTO '91*.

[Kur02] K. Kurosawa, "Multi-recipient public-key encryption with shortened ciphertext," in *PKC '02*. [Online]. Available: https://link.springer.com/content/pdf/10.1007/3-540-45664-3_4.pdf.

[LAZ+24] R. Linus, L. Aumayr, A. Zamyatin, A. Pelosi, Z. Avarikioti, and M. Maffei, *BitVM2: Bridging bitcoin to second layers*, https://bitvm.org/bitvm_bridge.pdf.

[LLX07] J. Li, N. Li, and R. Xue, "Universal accumulators with efficient nonmembership proofs," in *ACNS '07*. [Online]. Available: https://link.springer.com/content/pdf/10.1007/978-3-540-72738-5_17.pdf.

[LP07] Y. Lindell and B. Pinkas, "An efficient protocol for secure two-party computation in the presence of malicious adversaries," in *EUROCRYPT '07*. [Online]. Available: https://eprint.iacr.org/2008/049.pdf.

[LR14] Y. Lindell and B. Riva, "Cut-and-choose Yao-based secure computation in the online/offline and batch settings," in *CRYPTO '14*. [Online]. Available: https://eprint.iacr.org/2014/667.pdf.

[LWYY25a] H. Liu, X. Wang, K. Yang, and Y. Yu, "Authenticated BitGC for actively secure rate-one 2PC," in *CRYPTO '25*. [Online]. Available: https://eprint.iacr.org/2025/232.pdf.

[LWYY25b] H. Liu, X. Wang, K. Yang, and Y. Yu, "BitGC: Garbled circuits with 1 bit per gate," in *EUROCRYPT '25*. [Online]. Available: https://eprint.iacr.org/2024/1988.pdf.

[Lam79] L. Lamport, "Constructing digital signatures from a one way function," in *SRI International CSL '98*.

[Lin13] Y. Lindell, "Fast cut-and-choose based protocols for malicious and covert adversaries," in *CRYPTO '13*. [Online]. Available: https://eprint.iacr.org/2013/079.pdf.

[Lin23] R. Linus, *BitVM: Compute anything on Bitcoin*, https://bitvm.org/bitvm.pdf.

[Lin25a] R. Linus, *BitVM 3s: Garbled circuits for efficient computation on Bitcoin*, https://bitvm.org/bitvm3.pdf.

[Lin25b] R. Linus, *BitVM3-RSA: Efficient computation on Bitcoin bridges*, https://bitvm.org/bitvm3-rsa.pdf.

[Lip24] H. Lipmaa, "Polymath: Groth16 is not the limit," in *CRYPTO '24*. [Online]. Available: https://eprint.iacr.org/2024/916.pdf.

[MF06] P. Mohassel and M. Franklin, "Efficiency tradeoffs for malicious two-party computation," in *PKC '06*. [Online]. Available: https://www.iacr.org/archive/pkc2006/39580468/39580468.pdf.

[MORS25] P. Meyer, C. Orlandi, L. Roy, and P. Scholl, "Silent circuit relinearisation: Sublinear-size (Boolean and arithmetic) garbled circuits from DCR," in *CRYPTO '25*. [Online]. Available: https://eprint.iacr.org/2025/245.pdf.

[MP12] D. Micciancio and C. Peikert, "Trapdoors for lattices: Simpler, tighter, faster, smaller," in *EUROCRYPT '12*. [Online]. Available: https://eprint.iacr.org/2011/501.pdf.

[MR17] P. Mohassel and M. Rosulek, "Non-interactive secure 2PC in the offline/online and batch settings," in *EUROCRYPT '17*. [Online]. Available: https://eprint.iacr.org/2017/125.pdf.

[MS13] P. Mohassel and S. Sadeghian, "How to hide circuits in MPC: An efficient framework for private function evaluation," in *EUROCRYPT '13*. [Online]. Available: https://eprint.iacr.org/2013/137.

[Max11] G. Maxwell, *Zero knowledge contingent payment*, https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment.

[Mer87] R. C. Merkle, "A digital signature based on a conventional encryption function," in *CRYPTO '87*. [Online]. Available: https://link.springer.com/content/pdf/10.1007/3-540-48184-2_32.pdf.

[Mer89] R. C. Merkle, "A certified digital signature," in *CRYPTO '89*. [Online]. Available: https://link.springer.com/content/pdf/10.1007/0-387-34805-0_21.pdf.

[NO09] J. B. Nielsen and C. Orlandi, "LEGO for two party secure computation," in *TCC '09*. [Online]. Available: https://eprint.iacr.org/2008/427.pdf.

[NP01] M. Naor and B. Pinkas, "Efficient oblivious transfer protocols," in *SODA '01*.

[Ngu05] L. Nguyen, "Accumulators from bilinear pairings and applications to ID-based ring signatures and group membership revocation," in *CT-RSA '05*. [Online]. Available: https://eprint.iacr.org/2005/123.pdf.

[OSY21] C. Orlandi, P. Scholl, and S. Yakoubov, "The rise of paillier: Homomorphic secret sharing and public-key silent ot," in *EUROCRYPT '21*. [Online]. Available: https://eprint.iacr.org/2021/262.pdf.

[Orr25] M. Orrù, "Revisiting keyed-verification anonymous credentials," in *CCS '2025*. [Online]. Available: https://eprint.iacr.org/2024/1552.pdf.

[PRV12] B. Parno, M. Raykova, and V. Vaikuntanathan, "How to delegate and verify in public: Verifiable computation from attribute-based encryption," in *TCC '12*. [Online]. Available: https://eprint.iacr.org/2011/597.pdf.

[Pai99] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT '99*. [Online]. Available: https://link.springer.com/content/pdf/10.1007/3-540-48910-X_16.pdf.

[Pin03] B. Pinkas, "Fair secure two-party computation," in *EUROCRYPT '03*. [Online]. Available: https://iacr.org/archive/eurocrypt2003/26560087/26560087.pdf.

[Pol24] Polyhedra, *Expander, still the world's fastest ZK prover*, https://blog.polyhedra.network/expander-still-the-worlds-fastest-zk-prover/.

[Por22] T. Pornin, "Efficient and complete formulas for binary curves," in *ePrint 2022/1325*. [Online]. Available: https://eprint.iacr.org/2022/1325.pdf.

[Por23] T. Pornin, "Faster complete formulas for the GLS254 binary curve," in *ePrint 2023/1688*. [Online]. Available: https://eprint.iacr.org/2023/1688.pdf.

[QWW18] W. Quach, H. Wee, and D. Wichs, "Laconic function evaluation and applications," in *FOCS '18*. [Online]. Available: https://eprint.iacr.org/2018/409.pdf.

[RR21] M. Rosulek and L. Roy, "Three halves make a whole? Beating the half-gates lower bound for garbled circuits," in *CRYPTO '21*. [Online]. Available: https://eprint.iacr.org/2021/749.pdf.

[RS21] L. Roy and J. Singh, "Large message homomorphic secret sharing from DCR and applications," in *CRYPTO '21*. [Online]. Available: https://eprint.iacr.org/2021/274.pdf.

[RT17] P. Rindal and R. Trifiletti, "SplitCommit: Implementing and analyzing homomorphic UC commitments," in *ePrint 2017/407*. [Online]. Available: https://eprint.iacr.org/2017/407.pdf.

[Reg05] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *STOC '05*. [Online]. Available: https://arxiv.org/abs/2401.03703.

[Rub25a] J. Rubin, *Delbrag*, https://rubin.io/public/pdfs/delbrag.pdf.

[Rub25b] J. Rubin, *Delbrag*, https://rubin.io/public/pdfs/delbrag-talk-btcpp-austin-2025.pdf.

[SSE+24] R. Steinfeld, A. Sakzad, M. F. Esgin, V. Kuchta, M. Yassi, and R. K. Zhao, "LUNA: Quasi-optimally succinct designated-verifier zero-knowledge arguments from lattices," in *CCS '24*. [Online]. Available: https://eprint.iacr.org/2022/1690.pdf.

[SSS+22] H. Sun, H. Sun, K. Singh, A. S. Peddireddy, H. Patil, J. Liu, and W. Chen, "The inspection model for zero-knowledge proofs and efficient Zerocash with secp256k1 keys," in *ePrint 2022/1079*. [Online]. Available: https://eprint.iacr.org/2022/1079.

[Sch80] J. T. Schwartz, "Fast probabilistic algorithms for verification of polynomial identities," in *JACM '80*. [Online]. Available: https://dl.acm.org/doi/10.1145/322217.322225.

[Sch89] C.-P. Schnorr, "Efficient identification and signatures for smart cards," in *CRYPTO '89*. [Online]. Available: https://link.springer.com/content/pdf/10.1007/0-387-34805-0_22.pdf.

[Sha79] A. Shamir, "How to share a secret," in *CACM '79*. [Online]. Available: https://dl.acm.org/doi/10.1145/359168.359176.

[Sho97] V. Shoup, "Lower bounds for discrete logarithms and related problems," in *EUROCRYPT '97*. [Online]. Available: https://link.springer.com/content/pdf/10.1007/3-540-69053-0_18.pdf.

[Sli] *Slipstream*. [Online]. Available: https://slipstream.mara.com/.

[Spi95] D. A. Spielman, "Linear-time encodable and decodable error-correcting codes," in *STOC '95*. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/225058.225165.

[Sta24] StarkWare, *StarkWare sets new proving record*, https://starkware.co/blog/starkware-new-proving-record/.

[Tie13] TierNolan, *Re: Alt chains and atomic transfers*, https://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949.

[Tse25] D. Tse, *BitVM mainnet experiment*. [Online]. Available: https://x.com/dntse/status/1930272316124287393/.

[Val76] L. G. Valiant, "Universal circuits," in *STOC '76*. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/800113.803649.

[WOS+25] A. P. Y. Woo, A. Ozdemir, C. Sharp, T. Pornin, and P. Grubbs, "Efficient proofs of possession for legacy signatures," in *S&P '25*. [Online]. Available: https://eprint.iacr.org/2025/538.

[WRK17] X. Wang, S. Ranellucci, and J. Katz, "Authenticated garbling and efficient maliciously secure two-party computation," in *CCS '17*. [Online]. Available: https://eprint.iacr.org/2017/030.pdf.

[XZS22] T. Xie, Y. Zhang, and D. Song, "Orion: Zero knowledge proof with linear prover time," in *CRYPTO '22*. [Online]. Available: https://eprint.iacr.org/2022/1010.pdf.

[Yao86] A. C.-C. Yao, "How to generate and exchange secrets," in *FOCS '86*.

[ZHKs16] R. Zhu, Y. Huang, J. Katz, and a. shelat, "The cut-and-choose game and its application to cryptographic protocols," in *USENIX Security '16*. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_zhu.pdf.

[ZRE15] S. Zahur, M. Rosulek, and D. Evans, "Two halves make a whole: Reducing data transfer in garbled circuits using half gates," in *EUROCRYPT '15*. [Online]. Available: https://eprint.iacr.org/2014/756.pdf.

[Zip79] R. Zippel, "Probabilistic algorithms for sparse polynomials," in *EUROSAM '79*.

A Succinct PFGC from HMAC

In this section, we rephrase and summarize the instantiations of BitVM from HMAC in [ILL25b]. We then discuss succinctness and the security mechanisms to prevent a malicious garbler.

A.1 HMAC with unbounded depth from Paillier

[ILL25b] presents a HMAC construction relying on KDM-DCR security of the Paillier group, where KDM stands for key-dependent message and DCR stands for the decisional composite residuosity assumption. This assumption allows us to encrypt the secret key of Paillier encryption (more specifically, the inverse of the secret key) inside the same encryption.

The HMAC is based on Paillier public-key encryption [Pai99; DJ01] as follows.

- $\mathsf{KeyGen}(1^\lambda) \to (\mathsf{sk}, \mathsf{pk})$: The key generation algorithm is a randomized algorithm that takes the security parameters as input and generates a pair of keys.
  - sample suitable prime p and q and form $N = pq$
  - output $\mathsf{sk} = (p-1)(q-1)$ and $\mathsf{pk} = N$
- $\mathsf{Enc}(\mathsf{pk}, m) \to c$: The encryption algorithm is a randomized algorithm that takes the public key pk and a message $m \in \mathbb{Z}_N^+$ as input and generates the ciphertext c.
  - sample $r \leftarrow \mathbb{Z}_{N^2}^*$
  - output $c = r^N(1+mN) \pmod{N^2}$
- $\mathsf{Dec}(\mathsf{sk}, c) \to m$: The decryption algorithm is a deterministic algorithm that takes the secret key sk and a ciphertext $c \in \mathbb{Z}_{N^2}^*$ as input and outputs the plaintext m.
  - compute $t = c^{\mathsf{sk}} = 1 + m \cdot \mathsf{sk} \cdot N \pmod{N^2}$
  - compute $m = \frac{t-1}{N \cdot \mathsf{sk}} \pmod{N}$

An important result, by Roy and Singh [RS21], and also independently by Orlandi, Scholl, and Yakoubov [OSY21], shows that there is a so-called distance function on the Paillier group. Given a ciphertext that encrypts m called c, the distance between $c^{\mathsf{sk} \cdot x + z}$ and c^z is $\mathsf{sk} \cdot x \cdot m$ and can be calculated with an efficient function $\mathsf{Dist}(\cdot)$.

$$\mathsf{Dist}(c^{\mathsf{sk} \cdot x + z}) - \mathsf{Dist}(c^z) = \mathsf{sk} \cdot x \cdot m \pmod{N}$$

which is defined as follows for the Paillier group.

$$\mathsf{Dist}(c) = \frac{\left(\frac{c}{c \bmod N}\right) - 1}{N} \pmod{N}$$

More specifically, we only consider c that encrypts sk^{-1} (which is safe to do assuming the KDM-DCR security of the Paillier group), in which case:

$$\mathsf{Dist}(c^{\mathsf{sk} \cdot x + z}) - \mathsf{Dist}(c^z) = x \pmod{N}$$

Construction. It is now natural to present the overall construction.

- $\mathsf{KeyGen}(1^\lambda) \to \mathsf{sk}, \mathsf{evk}$: The key generation algorithm samples a Paillier key pair and computes the encryption of sk^{-1}.
  - $(\mathsf{sk}_{\mathsf{Pai}}, \mathsf{pk}_{\mathsf{Pai}}) \leftarrow \mathsf{Paillier.KeyGen}(1^\lambda)$
  - $c \leftarrow \mathsf{Paillier.Enc}(\mathsf{pk}_{\mathsf{Pai}}, \mathsf{sk}_{\mathsf{Pai}}^{-1})$
  - output $\mathsf{sk} = (\mathsf{sk}_{\mathsf{Pai}}, c)$ and $\mathsf{evk} = (\mathsf{pk}_{\mathsf{Pai}}, c)$
- $\mathsf{AuthAndEvalKey}(\mathsf{sk}) \to \{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}, L_o^{(0)}, L_o^{(1)}$: The authentication and evaluation-over key algorithm generates K_w for any input wire w and all the input and output labels.
  - parse $\mathsf{sk} = (\mathsf{sk}_{\mathsf{Pai}}, c)$
  - for $i \in [\ell]$, sample $K_i \in \mathbb{Z}_N^+$
  - for $i \in [\ell]$, compute $L_i^{(0)} = K_i \in \mathbb{Z}_N^+$ and $L_i^{(1)} = \mathsf{sk}_{\mathsf{Pai}} + K_i \in \mathbb{Z}_N^+$
  - for an addition gate that takes wires w_a and w_b as input and outputs the sum on w_c:
    $$K_{w_c} = K_{w_a} + K_{w_b}$$
    for a multiplication gate that takes wires w_a and w_b as input and outputs the product on w_c:
    $$K_{w_c} = -\mathsf{Dist}(c^{K_{w_a} \cdot K_{w_b}})$$
  - compute $L_o^{(0)} = K_o$ and $L_o^{(1)} = \mathsf{sk}_{\mathsf{Pai}} + K_o$, where K_o corresponds to the output wire
  - output $\{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}, L_o^{(0)}, L_o^{(1)}$
- $\mathsf{EvalTag}(\mathsf{evk}, (\mathbf{x}, \{L_i^{(x_i)}\}_{i\in[\ell]})) \to L_o^{(b_o)}$: The evaluation-over-tag algorithm allows an evaluator to figure out $L_o^{(b_o)}$ if $C(\mathbf{x}) = b_o$. It evaluates the circuit in the HMAC of the inputs gate-by-gate.
  - for an addition gate that takes wires w_a and w_b as input and outputs the sum on w_c:
    $$L_{w_c} = L_{w_a} + L_{w_b}$$
    for a multiplication gate that takes wires w_a and w_b as input and outputs the product on w_c:
    $$L_{w_c} = -\mathsf{Dist}(c^{L_{w_a} \cdot L_{w_b}}) + L_{w_a} \cdot b_{w_b} + L_{w_b} \cdot b_{w_a}$$
  - output $L_o^{(b_o)} = L_o$.

Optimization. The compute overhead of this HMAC is dominated by computing $c^{L_{w_a} \cdot L_{w_b}} \pmod{N^2}$ with N about 2048 bits. There are ways to speed up the computation.

- **Garbler:**
  - Since the garbler knows p and q, it can use the Chinese Remainder Theorem (CRT) to reduce the exponentiation to $L_{w_a} \cdot L_{w_b}$ into the two following exponentiations, with a smaller base ($N^2 \to p^2$ or q^2) and a smaller exponent ($\phi(N^2) \to \phi(p^2)$ or $\phi(q^2)$).
    $$(c \bmod p^2)^{L_{w_a} \cdot L_{w_b} \bmod \phi(p^2)} \pmod{p^2}$$
    $$(c \bmod q^2)^{L_{w_a} \cdot L_{w_b} \bmod \phi(q^2)} \pmod{q^2}$$

- The garbler can precompute lookup tables for powers of $(c \bmod p^2)^x \pmod{p^2}$ and powers of $(c \bmod q^2)^x \pmod{q^2}$.
- Modular multiplication over p^2 (or q^2) can be made slightly faster by decomposing them into the form of $ap + b$ (or $aq + b$) and observing how some terms are canceled.

- **Evaluator:**
  - Though the evaluator does not know p or q, it can precompute a lookup table for $c^x \pmod{N^2}$.
  - Modular multiplication over N^2 can be made slightly faster by decomposing them into the form of $aN + b$ and observing how some terms are canceled.

A.2 HMAC with unbounded depth from CP-RLWE

[ILL25a] presents another HMAC based on the circular-power RingLWE assumption (CP-RLWE). The idea is to provide additional ciphertexts to help with multiplication and use rounding to reduce noise in the middle of the computation. Fix a suitable polynomial ring $\mathcal{R} = \mathbb{Z}[X]/(X^n + 1)$ with n being a power-of-two, two modulus p and q satisfying certain requirements outlined in [ILL25a], and error and secret distributions $\mathcal{D}_{\mathsf{err}}$ and $\mathcal{D}_{\mathsf{sk}}$ with bounded coefficients.

- $\mathsf{KeyGen}(1^\lambda) \to \mathsf{sk}, \mathsf{evk}$: The key generation algorithm samples the necessary ring elements and computes the encryption of the secret s.
  - $s \leftarrow \mathcal{D}_{\mathsf{sk}}$
  - $a \leftarrow \mathcal{R}_q$ which can be generated from a transparent setup
  - $e_1, e_2 \leftarrow \mathcal{D}_{\mathsf{err}}$
  - $\mathsf{ct}_s = (a, s \cdot a + e_1, s^2 \cdot a + e_2 - s \cdot M)$ where M is a rounding parameter
  - generate a PRF key k
  - output $\mathsf{sk} = (a, s, k)$ and $\mathsf{evk} = (\mathsf{ct}_s, k)$
- $\mathsf{AuthAndEvalKey}(\mathsf{sk}) \to \{L_i^{(0)}, L_i^{(1)}\}_{i \in [\ell]}, L_o^{(0)}, L_o^{(1)}$: The authentication and evaluation-over key algorithm generates K_w for any input wire w and all the input and output labels.
  - parse $sk = (a, s, k)$
  - for $i \in [\ell]$, sample $K_i \in \mathcal{R}_p$ (importantly, not $\mathcal{R}_q$)
  - for $i \in [\ell]$, compute $L_i^{(0)} = K_i \in \mathcal{R}$ and $L_i^{(1)} = s + K_i \in \mathcal{R}$
  - for an addition gate that takes wires w_a and w_b as input and outputs the sum on w_c:

    $$K_{w_c} = K_{w_a} + K_{w_b}$$

    for a multiplication gate that takes wires w_a and w_b as input and outputs the product on w_c:

    $$K_{w_c} = \left\lfloor \frac{a \cdot K_{w_a} \cdot K_{w_b}}{M} \right\rceil + \mathsf{PRF}(k; w_c)$$

    where it is important that the output of PRF is in $\mathcal{R}_p$, not $\mathcal{R}_q$
  - compute $L_o^{(0)} = K_o$ and $L_o^{(1)} = s + K_o$, where K_o corresponds to the output wire

– output $\{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}, L_o^{(0)}, L_o^{(1)}$

- $\mathsf{EvalTag}(\mathsf{evk}, (\mathbf{x}, \{L_i^{(x_i)}\}_{i\in[\ell]})) \to L_o^{(b_o)}$: The evaluation-over-tag algorithm allows an evaluator to figure out $L_o^{(b_o)}$ if $C(\mathbf{x}) = b_o$. It evaluates the circuit in the HMAC of the inputs gate-by-gate.
  – for an addition gate that takes wires w_a and w_b as input and outputs the sum on w_c:

$$L_{w_c} = L_{w_a} + L_{w_b}$$

  for a multiplication gate that takes wires w_a and w_b as input and outputs the product on w_c. Parse $\mathsf{ct}_s = (a, b, c)$

$$L_{w_c} = \left\lfloor \frac{a \cdot L_{w_a} \cdot L_{w_b} - b \cdot (L_{w_a} \cdot b_{w_b} + L_{w_b} \cdot b_{w_a}) - c \cdot b_{w_a} \cdot b_{w_b}}{M} \right\rceil + \mathsf{PRF}(k; w_c)$$

  – output $L_o^{(b_o)} = L_o$.

Computation. Note that K_{w_a} and K_{w_b} have small coefficients (a few bits more than p), the same for L_{w_a} and L_{w_b} as s is small. The ring we use is communicative. So, when computing $a \cdot K_{w_a} \cdot K_{w_b}$ or $a \cdot L_{w_a} \cdot L_{w_b}$, one can first compute $K_{w_a} \cdot K_{w_b}$ or $L_{w_a} \cdot L_{w_b}$.

[ILL25a] uses RLWE parameters with q of length about 142 bits and $n = 8192$ for 128-bit security. The main computation is all about multiplying ring elements. Note that the label of each gate is a ring element of size 142KB, in contrast to the HMAC based on Paillier where each label is approximately 256B, so memory management is important, and one needs to remove inactive labels (labels that will not be used anymore) from memory in time.

Malicious security. To prevent a malicious garbler from generating incorrect labels, we need to verify the correctness of $\mathsf{HMAC.AuthAndEvalKey}(\mathsf{sk})$ (and the published hashes). NIZK can work but we feel that it might be too expensive, while C&C is practical. One can C&C the entire circuit or use DUPLO on HMAC (Appendix A.3). The standard C&C and the batched C&C for the full circuits naturally apply to HMAC.

A.3 DUPLO for HMAC

We first provide the background on DUPLO and then discuss how to generalize the technique in DUPLO [KNR+17] to HMAC in [ILL25b] based on Paillier (the KDM-DCR assumption) and RLWE (the circular-power RingLWE assumption).

Background: DUPLO for PFGC in [ZRE15]. Recall that the idea of C&C is to have the garbler generate s copies of the circuits, reveal s_1 of them for the evaluator to verify whether they were generated correctly, and keep the remaining $s_2 = s - s_1$ of them, assuming that at least one of the s_2 circuits is good. Such C&C deals with the circuit as a whole.

DUPLO generalizes C&C to repeating subcomponents of GCs and to apply C&C over them. Then, the remaining s_2 copies of the same subcomponent are soldered together and connected to other subcomponents of a circuit. DUPLO shows how to do the soldering and the connecting.

In [ZRE15], as in other free-XOR [KS08] garbling schemes, each label has $L_i^{(1)} = L_i^{(0)} \oplus \Delta$ where $\Delta \in \{0,1\}^\lambda$ is a global offset that applies to all gates in the same circuit and is a secret value that the garbler cannot leak to the evaluator. When we open the s_1 copies of the circuits (or subcomponents in the case of DUPLO), we have to leak their corresponding Δ. As a result, it requires C&C to use different Δ in each of the s copies.

This becomes a problem for soldering and connecting when we deal with the s_2 copies of the remaining circuits (or subcomponents) because, for in the j-th copy, the i-th label is of the form $L_{j,i}^{(b_i)} = L_{j,i}^{(0)} \oplus b_i \Delta_j$. Not only does the 0-value label $L_{j,i}^{(0)}$ vary between the s_2 copies, the global offset Δ_j also varies.

The idea is to have the garbler create XOR-homomorphic commitments for $L_{j,i}^{(0)}$ (only for those labels that need to be connected) and Δ_j. For the s_1 opened circuits, the evaluator also checks if the garbler made the commitments correctly. Later, the garbler can solder circuit j and $j+1$ by opening the commitments on $L_{j,i}^{(0)} \oplus L_{j+1,i}^{(0)}$ and $\Delta_j \oplus \Delta_{j+1}$.[15]

There are many ways to instantiate XOR-homomorphic commitments, but we want it to be publicly verifiable and non-interactive. [MR17] provides ways to instantiate such a scheme, including those of [LR14] or [FJNT16], without interactions through the Fiat-Shamir transform. If we use the Fiat-Shamir transform, we recommend reusing Bitcoin PoW for better choices of security parameters and performance. [RT17] provides an implementation of the XOR homomorphic scheme in [FJNT16].

DUPLO for [ILL25b]. We now generalize the technique to the HMAC construction in [ILL25b]. [ILL25b] provides a number of instantiations of HMAC with the following structure: for a computation result $y = f(x)$, the HMAC tag for y is of the following format:

$$L_y = \Delta \cdot y + K_y$$

where Δ is a global secret that does not need to be revealed during the cut-and-choose protocol. For Paillier-based HMAC, $\Delta = \mathsf{sk}_{\mathsf{Pai}}$. For CP-RLWE-based HMAC, $\Delta = s$. K_y is the HMAC key corresponding to the computation.

Remember that the cut and choose protocol is applied to the calculation of K_y in aHMAC, which must be repeated for s times. The only difference between these s times is the HMAC key appearing in the input labels. Now, after C&C, we have s_2 remaining HMAC keys $K_{j,y}$ for each output y. When the evaluator evaluates circuits j and $j+1$, it gets two tags for the same value y:

$$L_{j,y} = \Delta \cdot y + K_{j,y}$$
$$L_{j+1,y} = \Delta \cdot y + K_{j+1,y}$$

One can see that the only difference is the HMAC key. If we are doing C&C on the whole circuits rather than DUPLO, we do not have to solder the circuits, and we can simply change the Bitcoin script to allow challenging any one of the s_2 circuits. But, if we are doing DUPLO, we do need to solder them into the same key. This is done as follows. The garbler makes an additive

[15] DUPLO for PFGC is simpler than standard GC because we do not need to hide the "color bits".

homomorphic commitment (not XOR homomorphic) for each $K_{j,y}$. For the s_1 opened copies, the evaluator checks if the commitments were made correctly. For the s_2 remaining copies, the garbler opens the commitment of $K_{j,y} - K_{j+1,y}$ to reveal $K_{j,y} - K_{j+1,y}$ which allows the evaluator to convert between $L_{j,y}$ and $L_{j+1,y}$.

Performance. Note that DUPLO for HMAC mostly helps with the compute overhead. It also reduces communication overhead, but since the instantiations from HMAC are already succinct, the communication overhead itself is not a bottleneck.

B Reusable PFGC from AB-LFE

In this section, we rephrase and summarize the instantiations of BitVM from AB-LFE in [HLL23] and in [QWW18] in the modified syntax presented in § 3.3, in which AB-LFE provides all input labels at once. We start with [QWW18] since it is simpler and lays the foundation for [HLL23].

B.1 AB-LFE with bounded depth from LWE

[QWW18] presents an instantiation of AB-LFE largely based on the KP-ABE construction from [BGG+14] but with some simplifications. The construction is based on learning with errors (LWE). Fix suitable parameters n, q, χ, B from the LWE assumption and set $m = n\lceil \log q \rceil$. As follows, we use $\mathbf{G} \in \mathbb{Z}_q^{n \times m}$, which is a specific gadget matrix [MP12].

- $\mathsf{ursGen}(1^\lambda, \mathsf{params}) \to \mathsf{urs}$: The URS consists of ℓ random matrices, where ℓ is the length of the input to the circuit $C(\mathbf{x})$ and a digest of the circuit digest_C.
  - sample ℓ random matrices $\mathbf{A}_i \in \mathbb{Z}_q^{n \times m}$
  - for an addition gate that takes wires w_a and w_b as input and outputs the sum on w_c:

    $$\mathbf{A}_{w_c} = \mathbf{A}_{w_a} + \mathbf{A}_{w_b}$$

  - for a multiplication gate that takes wires w_a and w_b as input and outputs the product on w_c:

    $$\mathbf{A}_{w_c} = \mathbf{A}_{w_b} \cdot \mathsf{BD}(-\mathbf{A}_{w_a})$$

    where $\mathsf{BD}(\mathbf{A}) : \mathbb{Z}_q^{n \times m} \to \{0,1\}^{m \times m}$ replaces each element in $\mathbf{A}$ with a column vector $\mathbb{Z}_q^{\lceil \log q \rceil}$ which is the bit decomposition of that element from LSB to MSB (note that $m = n\lceil \log q \rceil$). This is for noise growth control. It is possible to swap $\mathbf{A}_{w_a}$ and $\mathbf{A}_{w_b}$, but it changes how noise grows, which we will discuss later.
  - set the digest of the circuit digest_C to be the matrix associated with the output wire, say $\mathbf{A}_o$
  - output $\mathsf{urs} = (\mathbf{A}_1, ..., \mathbf{A}_\ell, \mathbf{A}_o)$
- $\mathsf{Enc}(\mathsf{urs}, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1) \to \{L_i^{(0)}, L_i^{(1)}\}_{i \in [\ell]}, \mathsf{ct}_0, \mathsf{ct}_1$: The encryption algorithm outputs ℓ pairs of input labels and encrypts $\boldsymbol{\mu}_0 \in \{0,1\}^\ell$ under $\mathbf{A}_o$ and $\boldsymbol{\mu}_1 \in \{0,1\}^\ell$ under $\mathbf{A}_o + \mathbf{G}$. Assume $m \geq \lambda$.
  - sample $\mathbf{s} \leftarrow \mathbb{Z}_q^n$ and $\mathbf{e}_{i,0} \leftarrow \chi^m$, $\mathbf{e}_{i,1} \leftarrow \chi^m$ for $i \in [\ell]$
  - compute $L_i^{(0)} = \mathbf{s}^\top \mathbf{A}_i + \mathbf{e}_{i,0}^\top$ and $L_i^{(1)} = \mathbf{s}^\top (\mathbf{A}_i - \mathbf{G}) + \mathbf{e}_{i,1}^\top$, for all $i \in [\ell]$
  - sample $\mathbf{U}_0, \mathbf{U}_1 \leftarrow \mathbb{Z}_q^{n \times \ell}$ and decompose each element into bits from LSB to MSB, resulting in two bit matrices $\mathbf{T}_0, \mathbf{T}_1 \in \{0,1\}^{m \times \ell}$ such that $\mathbf{G} \cdot \mathbf{T}_0 = \mathbf{U}_0$ and $\mathbf{G} \cdot \mathbf{T}_1 = \mathbf{U}_1$
  - sample $\widetilde{\mathbf{e}}_0, \widetilde{\mathbf{e}}_1 \leftarrow [-B', B']^\ell$ where $B = (m+1)^d \cdot 2^\lambda \cdot B$, expected to be enough to noise-flood the output (we will later discuss the precise parameter for this)
  - compute $\boldsymbol{\beta}_0 = (\mathbf{s}^\top \mathbf{A}_o) \cdot \mathbf{T}_0 + \widetilde{\mathbf{e}}_0 + \boldsymbol{\mu}_0 \cdot \lfloor q/2 \rceil \in \mathbb{Z}_q^\ell$
  - compute $\boldsymbol{\beta}_1 = (\mathbf{s}^\top (\mathbf{A}_o + \mathbf{G})) \cdot \mathbf{T}_1 + \widetilde{\mathbf{e}}_1 + \boldsymbol{\mu}_1 \cdot \lfloor q/2 \rceil \in \mathbb{Z}_q^\ell$

- output $\{L_i^{(0)}, L_i^{(1)}\}_{i\in[\ell]}$, $\mathsf{ct}_0 = (\boldsymbol{\beta}_0, \mathbf{T}_0)$, and $\mathsf{ct}_1 = (\boldsymbol{\beta}_1, \mathbf{T}_1)$. Note that $\mathbf{T}_0$ and $\mathbf{T}_1$ can be compactly stored by using a PRNG to generate $\mathbf{U}_0$ and $\mathbf{U}_1$ and storing the PRNG seed instead.

- $\mathsf{Dec}(\mathsf{urs}, (\mathbf{x}, \{L_i^{(x_i)}\}_{i\in[\ell]}), \mathsf{ct}_0, \mathsf{ct}_1) \to \boldsymbol{\mu}_{b_o}$: The decryption algorithm evaluates the circuit gate-by-gate using the labels.
  - for an addition gate that takes wires w_a and w_b as input and outputs the sum on w_c:

$$L_{w_c} = L_{w_a} + L_{w_b}$$

  - for a multiplication gate that takes wires w_a and w_b as input and outputs the product on w_c:

$$L_{w_c} = b_{w_b} \cdot L_{w_a} + \mathsf{BD}(-\mathbf{A}_{w_a}) \cdot L_{w_b}$$

    where b_{w_b} is the value on wire w_b
  - recover $\boldsymbol{\mu}_{b_o} = \lfloor (\boldsymbol{\beta}_{b_o} - L_o \cdot \mathbf{T}_{b_o})/q \rceil$

Bounded depth. AB-LFE from [QWW18] works only for circuits under a pre-configured depth because, to work with decryption, one must ensure that the noise does not cause $\boldsymbol{\mu}$ to be unrecovered. For each multiplication, the output wire w_c carries a label with noise bounded by:

$$\|\mathbf{e}_{w_c}\| \leq \|\mathbf{e}_{w_a}\| + m \cdot \|\mathbf{e}_{w_b}\|$$

Many gates in a Boolean circuit require multiplication, including XOR gates. If a circuit has depth d, then the modulus q should be at least $(m+1)^d \cdot 2^\lambda$ larger than the initial noise (bounded by B). [QWW18] gives $q/B = 8 \cdot (m+1)^{d+1} \cdot 2^\lambda$ due to noise flooding and decryption. The compute overhead of AB-LFE from [QWW18] increases with $O(d^2)$.

The more accurate notion for depth is called *pebbling depth* [MORS25], which explores the asymmetry on how $\|\mathbf{e}_{w_a}\|$ and $\|\mathbf{e}_{w_b}\|$ contribute to $\|\mathbf{e}_{w_c}\|$ and swap w_a and w_b to minimize noise growth. This generalizes *polynomial gates* in [BGG+14]. If the pebbling depth is d', we can require $q/B = 8 \cdot (m+1)^{d'+1} \cdot 2^\lambda$ and set $B' = (m+1)^{d'} \cdot 2^\lambda \cdot B$ for noise flooding.

How to minimize the pebbling depth is a question on its own. If one does not care about the circuit size, one can use the disjunctive normal form (DNF) to represent C with ℓ bit input with $q/B = 2^{3+\ell+\lambda} \cdot \ell \cdot m^2$, but $|C|$ could be 2^ℓ times larger. Note that for BitVM with Groth16, $\ell = 1270$, and the circuit basically enumerates all valid Groth16 proofs under that verification key in the world, which would be unrealistic.

B.2 AB-LFE with unbounded depth from csLWE

[HLL23] presents an instantiation of AB-LFE under the circular small secret learning with errors (csLWE) assumption. Unlike [QWW18], the parameters of [HLL23] do not have to grow with the depth of the circuit because there is a way to suppress the noise in the middle of the computation. This construction is mostly theoretical, as noise suppression can be very heavy.

Necessary helper matrices. [HLL23] presents an algorithm that involves some helper matrices. Assume q to be a power of 2, M be a power of 2 smaller than q, and $m = 3(n+1)\lceil \log q \rceil$. There is

a gadget matrix $\mathbf{G} \in \mathbb{Z}_q^{(n+1)\times m}$, which replaces "1" in the identity matrix $\mathbf{I}_{n+1} \in \{0,1\}^{(n+1)\times(n+1)}$ with a row vector $(1, 2, 2^2, ..., 2^{\frac{m}{n+1}-1})$ and "0" with a row vector $0^{1\times 3\lceil \log q \rceil}$.

Let $M = 2^k$. We can derive two more matrices $\mathbf{G}_L \in \mathbb{Z}_q^{(n+1)\times(n+1)k}$ and $\mathbf{G}_R \in \mathbb{Z}_q^{(n+1)\times(n+1)(\frac{m}{n+1}-k)}$. $\mathbf{G}_L$ contains all nonzero entries smaller than M by replacing "1" in $\mathbf{I}_{n+1}$ with $(1, 2, ..., M/2)$. $\mathbf{G}_R$ contains all entries no less than M by replacing "1" with $\mathbf{I}_{n+1}$ with $(M, 2M, ..., 2^{\frac{m}{n+1}-1})$.

One can see that $(\mathbf{G}_L \mid \mathbf{G}_R) \in \mathbb{Z}_q^{(n+1)\times m}$ is the result of permuting the columns in $\mathbf{G}$. That is to say, there is a permutation matrix $\mathbf{Q} \in \{0,1\}^{m\times m}$ (a binary matrix that only contains a single "one" for each row and for each column) such that $(\mathbf{G}_L \mid \mathbf{G}_R) \cdot \mathbf{Q} = \mathbf{G}$.

Next, we also have the algorithm $\mathsf{BD}(\mathbf{A}) : \mathbb{Z}_q^{(n+1)\times n_c} \to \{0,1\}^{m\times n_c}$ for any n_c that replaces each element in the matrix with its bit decomposition as a row vector of length $3\lceil \log q \rceil$. We have $\mathbf{G} \cdot \mathsf{BD}(\mathbf{A}) = \mathbf{A}$.

Noise reduction. Consider a label with noise that we want to reduce.

$$L_w = \mathbf{s}^\top(\mathbf{A}_w - b_w\mathbf{G}) + \mathbf{e}_w \in \mathbb{Z}_q^m$$

[HLL23] gives an algorithm called $\mathsf{RemoveNoise}(\mathbf{u} \in \mathbb{Z}_q^m)$ that works as follows:

- compute $\mathbf{v}_L = \mathbf{u} \cdot \mathsf{BD}(M \cdot \mathbf{G}_L) \in \mathbb{Z}_q^{(n+1)k}$ and $\mathbf{v}_R = \mathbf{u} \cdot \mathsf{BD}(\mathbf{G}_R) \in \mathbb{Z}_q^{(n+1)(\frac{m}{n+1}-k)}$
- compute $\mathbf{w}$ as follows:

$$\mathbf{w} = \left(\left\lfloor \frac{\mathbf{v}_L \bmod q}{M} \right\rceil, \; M \cdot \left\lfloor \frac{\mathbf{v}_R \bmod q}{M} \right\rceil \right) \cdot \mathbf{Q}$$

We know: $\mathsf{RemoveNoise}(L_w) = \mathsf{RemoveNoise}(K_w) - b_w \cdot \mathbf{s}^\top \cdot \mathbf{G}$.

With high probability and proper parameters (M needs to be large enough to remove $\mathbf{e}_L$ and $\mathbf{e}_R$), almost all the noise has been removed. The only remaining problem is that $\mathsf{RemoveNoise}(L_w)$ no longer preserves the structure of a label in AB-LFE and cannot continue the computation.

We can restore it to a label under a new matrix $\mathbf{A}'_w$ if the evaluator has a way to know $\mathbf{s}^\top\mathbf{A}'_w - \mathsf{RemoveNoise}(K_w)$, which is a function on the secret $\mathbf{s}$. [HLL23] shows that this can be done in a bootstrapping-like protocol.

Bootstrapping. [HLL23] uses the dual-use technique in [BTVW17] that creates encryption of secrets $\mathbf{s}$ under GSW encryption [GSW13] using $\mathbf{s}$ as the secret key. The ciphertext is denoted by $\mathbf{S}$. One can create a Boolean circuit that takes secret $\mathbf{s}$ as input and computes $\mathsf{RemoveNoise}(K_w) = \mathsf{RemoveNoise}(\mathbf{s}^\top\mathbf{A}_w)$ and then run this circuit in AB-LFE over labels for each bit of $\mathbf{S}$ provided as additional input and rearrange the output bits as $\mathbb{Z}_q^m$. Due to [BTVW17], the AB-LFE label for the output is as follows.

$$L^* = \mathbf{s}^\top\mathbf{A}^* - \mathsf{RemoveNoise}(K_w) + \mathbf{e}^*$$

where $\mathbf{A}^*$ does not depend on $\mathbf{s}$ but depends on $\mathbf{A}_w$, and $\mathbf{e}_w$ grows with the depth of the circuit that represents the GSW homomorphic evaluation. Adding L^* to $\mathsf{RemoveNoise}(L_w)$ from the previous discussion gives us a new label with the same value in L_w, but under a new matrix $\mathbf{A}'_w = \mathbf{A}^*$.

$$\begin{aligned} L'_w &= \mathsf{RemoveNoise}(L_w) + L^* \\ &= \mathbf{s}^\top(\mathbf{A}^* - b_w \cdot \mathbf{G}) + \mathbf{e}^* \end{aligned}$$

Since noise does not depend on the depth of the original circuit, C, the parameters do not need to be set to accommodate that depth, but only what is necessary to ensure $\mathbf{e}_w$ is still small.

B.3 Malicious security

We now discuss how to achieve malicious security against the garbler. What needs to be verified is $\mathsf{Enc}(\mathsf{urs}, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1)$. If there is no verification, a malicious garbler can break the protocol by not encrypting $\boldsymbol{\mu}_0$ or $\boldsymbol{\mu}_1$ correctly.

There are two ways to prove the correctness: NIZK or C&C.

- **NIZK:** We can resort to general-purpose NIZK, with a few remarks, focusing on [QWW18] as an example:
  - A lot of the computation is multiplying (e.g. $\mathbf{A}_o\mathbf{T}_0$, $(\mathbf{A}_o + \mathbf{G})\mathbf{T}_0$) with a private value (e.g. $\mathbf{s}$).
  - Though [QWW18] samples $\mathbf{s} \leftarrow \mathbb{Z}_q^n$, in recent years it is fairly common to use small secret LWE (under a different assumption) like in [HLL23], in which case $\mathbf{s}$ would be small.
  - To verify in NIZK that the noise is generated correctly, it suffices to bound the Gaussian noise and check that the noise is within the bound.
- **C&C:** Note that $\mathsf{Enc}(\mathsf{urs}, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1)$ does not require a secret key, and its output depends only on the randomness used to generate it. The garbler can sample a few PRNG seeds and generate the corresponding outputs, and allow C&C to open and reveal some of the seeds for verification. This can be very efficient.

In either way, there is a need to compute hashes of $\boldsymbol{\mu}_0$ and $\boldsymbol{\mu}_1$ that are used to construct the BitVM transactions. NIZK or C&C is responsible for verifying that the hashes are computed correctly.

C Hash-based SNARK

A challenge in implementing BitVM is that the garbled circuit can be pretty large, which puts pressure both on the garbler and on the evaluator. This has to be with the inefficiency to represent elliptic curve computation in Boolean circuits in the SNARK verifier.

One alternative is to consider STARK, which might have a smaller Boolean circuit size, with other benefits. STARK can use much smaller fields (e.g. modulo $2^{31} - 1$ as in Circle STARK [HLP24]), and when it comes to hash functions, one can consider SHA-256 in Boolean circuits, each invocation of which takes 22272 AND gates, compared with 102093 AND gates for one single Fq multiplication under Montgomery representation. It is safe to assume that the majority of the overhead for verifying a STARK proof is about SHA-256, and the total number of gates should be orders of magnitude smaller than that of the SNARK proofs.

Nevertheless, STARK verification poses a unique challenge in that the proof is very long (usually hundreds of KB) compared to a SNARK proof (about 0.2KB). It would not be practical to reveal the label in a single Assert transaction. Instead, we would adopt the BitVM bridge [LAZ+24] design to have multiple Assert transactions, each of which is only revealed when challenged. There are efficient ways to manage a large number of Assert transactions, with the expectation that a challenger only needs to open a few Assert transactions to be able to narrow down which part of the computation is wrong.

The beauty of having multiple Assert transactions is that we can set enough checkpoints throughout the computation so that the evaluator does not have to challenge the full STARK verifier, but instead the entire computation can be divided into many small components, and the challenger only needs to challenge one of these components. One natural idea is to arrange each Merkle tree verification as a separate component, and another natural idea is that since each FRI query is independent, we can arrange them also in separate components (note that the challenger only needs to identify one query that fails the FRI test).

We can have separate garbled circuits for each of these components, with connectors created between them. The connectors are not by default revealed, but are only revealed on the relevant Assert transactions. This allows some of the garbled circuits to be used in a subsequent invocation if they were not used and their connectors were not revealed.

Some of these techniques and optimizations also naturally generalize to the SNARK verifier, where checkpoints can help reduce the amount of computation needed for challengers and may allow some garbled circuits to be used next time if they were not used and their connectors were not revealed, as the cost of having more inputs and a larger on-chain cost.

D Related work on recent BCIOP13 DV-SNARK

We also discuss some recent work on BCIOP13 DV-SNARK based on ElGamal encryption and why they did not address the major limitation related to the somewhat expensive discrete log that the verifier needs to perform.

Lookup table. [BIOW20] provides a way to reduce the overhead of the verifier for ElGamal groups so that, after a private preprocessing that generates a precomputation table, the verifier only needs to perform very few operations in addition to a table lookup. This avoids the need to perform discrete log, and if a malicious prover generates invalid proofs, it either falls outside the table or fails some equality tests, which can somehow be challenged in GC with some interactions between the challengers and operators that can happen on-chain. There are a few limitations:

- The sizes of the precomputation table and the CRS do not scale well with larger circuits.
- The preprocessing is private and can only be done by the challenger who plays the role of a verifier, and the operator may want the challenger to prove, using NIZK or cut-and-choose, that the table used for challenging is correct, which might be a significant cost.
- Implementing membership or non-membership proofs likely require Merkle trees [Mer87], RSA accumulators [BM93; LLX07; KL24], or bilinear accumulators [Ngu05; DT08], and some effort is needed to hide sensitive information from the garbler when requesting the input labels (which can be done with having additional input of a one-time pad and requesting the input labels of them beforehand using maliciously secure OT). Merkle trees require many more input bits similar to a STARK verifier discussed in Appendix C. RSA accumulators and bilinear accumulators are likely more expensive than a Groth16 verifier.

Distributed discrete-log based decryption. [ADI25] provides a way to compress the proof in ElGamal-based DV-SNARK by using a packed variant of ElGamal encryption (which can be referred to as Kurosawa encryption [Kur02]) and distributed discrete log (DDLog) [BGI16]. It is still necessary to use small groups and therefore the same proof needs to be repeated a few times for soundness amplification. Also, the program needs to be expressed as a Boolean circuit. However, it no longer requires a discrete log, but instead a DDLog over DDH groups, as follows from [BGI16]:

$$\mathsf{DDLog}(h) = \text{the smallest } y \quad \text{such that} \quad \phi(h \cdot g^y) = 0^\ell$$

where $\phi : \mathbb{G} \to \{0,1\}^\ell$ is a random function that can be instantiated with a random oracle, and y cannot be large and must be bounded. There are other hash functions [BGI17; BCG+17] but are similar (and will face the same issue).

DDLog can be used to compute the distance between h and $h \cdot g^\Delta$ without discrete log if Δ is not too large. However, there could be an error in which the calculated distance does not equal Δ but something else (hitting another value such that $\phi(h \cdot g^y) = 0^\ell$). This error can be problematic, because a valid proof would be considered invalid. [ADI25] has a solution. It asks the prover to randomize the proof until the prover can confirm that for each ciphertext element c in the proof, for all Δ in a certain bound $[-B, B]$,

$$\mathsf{DDLog}(c \cdot g^\Delta) = \mathsf{DDLog}(c) - \Delta$$

This can be shown to only require a small number of attempts under proper parameters. However, there is an issue related to DDLog.

Checking the verifier's computation of DDLog, especially in the GC, is difficult because y needs to be the *smallest* one that satisfies the requirement, which cannot be proven without showing that any $0 \leq y' < y$ does not satisfy the requirement. If we relax the requirement, do not require the "smallest", and only check $\phi(h \cdot g^y) = 0^\ell$ in the GC, there can be $y' \neq y$ that satisfies the requirement, which could be much larger than y but still within the search range. This becomes a problem when a malicious evaluator can use y' instead of y, even if y is smaller, and the GC would misclassify a valid proof as invalid (in standard BitVM), or an invalid proof as valid (in reverse BitVM, though this may only have a minor impact on security).

These issues are in essence the same as the original issue of [BCI+13] where a discrete log is needed for certain groups, and it is difficult for the verifier to compute this in GC.

E DV-SNARK vs multi-signature

It is important to compare DV-SNARK with traditional multi-signature because the security guarantees that they provide seem to be similar.

DV-SNARK requires one out of the n challengers (denoted by (n, n)) to be honest, while multi-signature can be configured with different threshold parameters, and can also support (n, n). The difference is on liveness guarantees against corrupted challengers with respect to a honest operator. We consider two types of corruption following the definitions in [BZ24]: active corruption and omission corruption.

- **active corruption:** This kind of corruption is often referred to as being malicious, means that the challengers can deviate from the protocol arbitrarily. In our case, it means that challengers may want to declare a valid proof as invalid or simply refuse to participate in the protocol.
- **omission corruption:** This kind of corruption is more about a challenger being not available and does not respond to the protocol. This represents the case where a challenger has network issues or is just being lazy.

Both protocols can preserve liveness under omission corruption.

- For DV-SNARK, it is straightforward. If the challenger was omission-corrupted before starting to challenge, then the operator does not need to commit the proof for this challenger because the challenger seems satisfied with off-chain evidence. If the challenger was omission-corrupted after starting the challenge protocol but before revealing the CRS, the challenger will time out, and BitVM will end the challenge protocol in favor of the operator. There is no further interaction required for the challenger.
- For multi-signature, we can implement (n, n) either by requiring n signatures in favor of the operator, or requiring no signatures against the operator. We consider only the latter. In that case, omission corruption does not produce any signatures against the operator.

The weakness of multi-signature is on active corruption.

- For DV-SNARK, it can tolerate that all n challengers are malicious and want to declare a valid proof invalid.
- For multi-signature, it fails as long as one of the challengers is malicious, who produces a signature against the honest prover.

This issue that malicious challengers can always abort the protocol is why multi-signature protocols often consider a majority vote where $t \approx n/2$ (or sometimes $t \approx 2n/3$) in the (t, n) threshold. This provides a weaker security guarantee than (n, n) against a malicious operator.

Note that both DV-SNARK and multi-signature are weaker than publicly verifiable SNARK, which is secure against active and omission corruption of public challengers.

Separately, it is important to also raise the issue of liveness requirements of the operator. BitVM naturally requires the operator to be online, while multi-signature naturally decentralizes the operator. It is possible to apply the same technique to BitVM for liveness, but it requires the operator's secret to be properly secret-shared among *an operator committee*. Otherwise, a malicious participant in the operator committee who has full knowledge of the operator's secret might be able to collude with a challenger to declare a valid proof invalid.

If we want to prevent such a situation for publicly verifiable SNARK, which the garbled circuit is generated by the operator, we may end up having to have the operator committee run the operator's algorithm in secure multiparty computation (MPC) [GMW87; BGW88], which is not very practical for BitVM from HMAC or from traditional privacy-free garbled circuit as the amount of MPC work grows with the size of the circuit.

However, if we focus on designated verifiers, decentralizing the operator could be made easier using reverse BitVM, which shifts the burden of generating the garbled circuits to the challengers, thereby keeping the operator's computation simple.