Delbrag

April 4, 2025

Jeremy Rubin
jeremy [at] char [dot] network

1 Disclaimer

This is published as a draft, sharing for the benefit of all in an unfinished state.

2 Introduction

Delbrag is a new improvement to BitVM style constructions that take advantage of Yao's Garbled Circuits to minimize the on-chain costs.

3 Construction of Encrypted NAND System with Zero-Knowledge Proof

In this section we give a brief demonstration of how a garbled circuit is built.

3.1 1. Represent the Inputs

The inputs are given as:

$$b_0 = [H(X_0), H(Y_0)], \quad b_1 = [H(X_1), H(Y_1)], \quad b_2 = [H(X_2), H(Y_2)]$$

where:

- X is a 0-bit value, and Y is a 1-bit value.
- H is a cryptographic hash function, e.g. SHA256.

The system must satisfy the following:

$$\text{NAND}(b_0, b_1) = b_2$$

Additionally, Y_2 must be encrypted under X_0 and X_1, allowing decryption when X_0 and X_1 are revealed:

$$K = H(X_0 \,||\, X_1)$$
$$\text{ENC}(K, Y_2) = c$$

2. Encryption of Y_2

To ensure that Y_2 is decryptable only with X_0 and X_1, we define:

$$K = H(X_0 \,||\, X_1)$$

as the derived encryption key. Using K, Y_2 is encrypted:

$$c = \text{Encrypt}(K, Y_2)$$

Decryption is only possible if both X_0 and X_1 are known, allowing the computation of K:

$$Y_2 = \text{Decrypt}(c, K).$$

3. Zero-Knowledge Proof of Correctness

To prove in Zero-Knowledge that knowing X_0 and X_1 allows decryption of Y_2, we can use a myriad of proof techniques. Later, we will discuss a practical construction.

Prover's Commitment: The prover commits to the values X_0, X_1, and Y_2 without revealing them:

$$C_0 = \text{Commit}(X_0), \quad C_1 = \text{Commit}(X_1), \quad C_2 = \text{Commit}(Y_2).$$

The prover computes the encryption:

$$c = \text{ENC}(H(X_0 \,||\, X_1), Y_2).$$

Proof: The prover demonstrates knowledge of X_0 and X_1 such that:

1. $K = H(X_0 \,||\, X_1)$,
2. $\text{Decrypt}(c, K) = Y_2$,
3. Y_2 satisfies $\text{NAND}(b_0, b_1) = b_2$.

This is achieved by encoding the above logic in a zk-SNARK circuit, which proves the following:

- K is derived correctly from X_0 and X_1,
- c decrypts to Y_2 with K,
- Y_2 satisfies the logical constraint $\text{NAND}(b_0, b_1) = b_2$.

The proof reveals no information about X_0, X_1, or Y_2, but convinces the verifier that the prover knows X_0 and X_1, and that c and Y_2 are correctly formed.

3.2 Verification

The verifier checks:

1. The zk-SNARK proof for validity,
2. The consistency of the provided b_0, b_1, b_2 with the proof and the NAND computation,

Summary

This construction ensures that:

1. Y_2 is only accessible if both X_0 and X_1 are known.
2. Proofs of correctness and knowledge are Zero-Knowledge.
3. The NAND operation $\text{NAND}(b_0, b_1) = b_2$ is verifiable.

4 Generalization

Commitments Per Gate

For each gate, such as gates $i = 0, j = 1, k = 2$, the full set of commitments is defined as:

Input 0	Input 1	Output	Encryption
X_0	X_1	Y_2	$H_{zk}(X_0 \| X_1) \bigoplus Y_2$
X_0	Y_1	Y_2	$H_{zk}(X_0 \| Y_1) \bigoplus Y_2$
Y_0	X_1	Y_2	$H_{zk}(Y_0 \| X_1) \bigoplus Y_2$
Y_0	Y_1	X_2	$H_{zk}(Y_0 \| Y_1) \bigoplus X_2$

This set includes all possible combinations of inputs X_0, X_1, Y_0, Y_1 that map to the output X_2, Y_2 for the respective gate.

The encryption is given as an example of a possible encryption function. Computing $H(a||b)$ requires knowledge of both a and b, and the xor operator obscures the output. H_{zk} could be SHA256, or a ZK hash that is easy to prove over such as Poseidon.

Commitments for a Full Circuit

For a complete circuit consisting of multiple gates, there will be many such sets of commitments:

$$\mathcal{C} = \bigcup_i C_i,$$

where i iterates over all gates in the circuit. Thus, the total commitments encompass all input-output relationships across all gates.

Aggregated Zero-Knowledge Proof

Rather than constructing a separate Zero-Knowledge Proof (ZKP) for each individual bit or gate, the ZKP can be aggregated across all gates and their respective commitments. This aggregated ZKP proves, in a single proof, that:

1. The prover knows the inputs for each gate.
2. The commitments and logical operations (e.g., NAND) for all gates in the circuit are valid.
3. The outputs of intermediate gates align with the inputs of subsequent gates.

This approach reduces the computational overhead of verifying many separate proofs and provides a compact, global proof for the correctness of the entire circuit.

Handling Equivocation on Interior Wires

If an interior wire e is equivocated over (e.g., its value is shared out of band), the verifier learns both X_e and Y_e. While this guarantees that the verifier can determine the correct result of the circuit, it introduces additional considerations:

1. Impact on the Verifier The verifier learning an additional wire's value is not inherently problematic, as they still obtain the correct circuit result.

2. Impact on the Prover This situation is problematic for the prover, as the verifier may use the extra information to construct a failure gate or otherwise compromise the prover's intent.

3. Mitigating Equivocation with Circuit Extensions To explicitly cause a failure in case of equivocation, an extension to the circuit can be implemented as follows:

1. **Encrypting an Equivocation Value:** Under each wire pair (X_i, Y_i), encrypt a single shared value X_{equiv} that is tied to the pair.
2. **Zero-Knowledge Proof Integration:** Incorporate into the Zero-Knowledge Proof a condition such that:
   - If any pair (X_i, Y_i) is revealed, X_{equiv} is also revealed.
   - The circuit is extended with a fail gate mechanism (see Section 5.0.2) to trigger a failure upon the revelation of X_{equiv}.

This mechanism ensures that any equivocation by the prover or unauthorized disclosure of wire values results in a clear and enforceable failure condition.

5 Integration into a Bitcoin Script

To integrate into Bitcoin, we introduce a 2 party prover-verifier model.

First, the prover and verifier generate a circuit and proof of NAND gate construction. The prover knows all of the underlying bit commitments. The prover has a key $\mathcal{P}$ and the verifier has a key $\mathcal{V}$.

We then filter the circuit to gates that are either input or output. I.e., the inputs are wires that are not an output to any gate, the outputs are wires that are not an input to any gate $input = \{i| \nexists \mathcal{C}_{.,.,i}\}$, $output = \{i| \nexists \mathcal{C}_{i,.,.} \vee \nexists \mathcal{C}_{.,i,.}\}$[1].

We will be accumulating the following script conditions into a taproot output ζ.

For each gate $b_k = NAND(b_i, b_j)$, we generate a set of 4 fraud proof gates for correctness of computation.

Each leaf will have the following form:

$$Incorrect(A, B, C) = AND(Signed(\mathcal{V}), \{Reveal(SHA256(X_i))|w \in A, B, C\})$$

And is templated in via the table below:

b_i	b_j	b_k	$\neg b_k$	$leaf$
X_i	X_j	Y_k	X_k	$Incorrect(X_i, X_j, X_k)$
Y_i	X_j	Y_k	X_k	$Incorrect(Y_i, X_j, X_k)$
X_i	Y_j	Y_k	X_k	$Incorrect(X_i, Y_j, X_k)$
Y_i	Y_j	X_k	Y_k	$Incorrect(Y_i, Y_j, Y_k)$

Thus, if there is any incorrect computation or revelation of an incorrect wire, ever, then the Verifier can sweep the funds.

Note: we do not need to create an exclusivity fraud proof tap-leaf, as we have already created X_{equiv}. However, if desired, a Tapleaf for each bit i could be included instead:

$$And(Reveal(SHA256(X_i)), Reveal(SHA256(Y_i)), Signed(\mathcal{V}))$$

Thus, if the verifier ever reveals multiple preimages for a given bit, the verifier may take all the funds.

5.0.1 Timeout

Should there be no bits revealed by deadline τ, then the Verifier can sweep the funds, or funds can be returned via a presigned return transaction for ζ.

Presigned:

$$And(After(\tau), Signed(\mathcal{P}), Signed(\mathcal{V}))$$

Sweep:

$$And(After(\tau), Signed(\mathcal{V}))$$

[1]Strictly speaking, a circuit might be set up such that there are interior wires desired to be also read as outputs. A practical implementation would likely use metadata to designate such outputs.

5.0.2 Failure Gates

Certain hash commitments can serve as indicators of invalid inputs. For instance, consider a circuit where b_p is set as follows:

$$b_p = \begin{cases} 1 & \text{if the input is prime,} \\ 0 & \text{if the input is not prime.} \end{cases}$$

If the input is required to be prime, a Tapleaf can be added to enforce conditions such that the commitment $\text{Sha256}(X_p)$, indicating the input was not prime, must be revealed to allow spending.

This can be expressed as:

$$\text{And}(\text{Signed}(\mathcal{V}), \text{Reveal}(\text{Sha256}(X_p))),$$

This construct ensures that the Tapleaf is spendable only if the verifier reveals evidence of an invalid (non-prime) input.

It is possibly more efficient in TapTree lookup cost to sum all of the possible failure conditions into a single bit, so as to reduce the number of Tapleafs.

However, if a failure Tapleaf instead includes a condition such as:

$$\text{And}(\text{Signed}(\mathcal{P}), \text{Signed}(\mathcal{V}), \text{Reveal}(\text{Sha256}(X_p))),$$

This setup allows for the generation of a pre-signed transaction associated with a specific case, such as when the commitment X_p reveals that the input is not prime. The pre-signed transaction ensures that spending can only occur under the following conditions:

1. The prover has explicitly signed for the transaction ($\mathcal{P}$).
2. The verifier has also signed for the transaction ($\mathcal{V}$).
3. The hash commitment $\text{Sha256}(X_p)$ is revealed, confirming the invalid (non-prime) nature of the input.

This construct tightly binds the transaction to the specific invalidity case, such as revealing a non-prime number, while ensuring dual authorization from both the prover and verifier.

5.1 Trace Delegate Input

Prover and Verifier agree that a separate input, δ, will be used to spend with the script built thus far. The input δ is a delegate input, similar to the concept of a connector.

Prover and verifier pre-sign a transaction (using a script clause as below).

$$And(Signed(\mathcal{P}), Signed(\mathcal{V}))$$

Prover signs this branch with **sighash_none**, but committing to δ and an nSequence with a timeout of ψ blocks [2]. The nSequence ensures that after δ's publication there is sufficient time for the proof of fraud to be used.

[2] BIP-341: The signature message commits to all input nSequence if SIGHASH_NONE or SIGHASH_SINGLE are set (unless SIGHASH_ANYONECANPAY is set as well).

5.2 δ Spend Conditions

The protocol's most expensive step space wise is the input step.

This step occurs in a separate input, γ, which needs only to contain a dust-amount of Bitcoin.

First, all the input wires must be filtered for, i.e., $I = \{i| \not\exists \mathcal{C}_{.,.,i}\}$

A trace of the input data must be committed to along with a spend by the verifier:

$$And(And(\forall i \in I : Or(Reveal(SHA256(X_i), SHA256(Y_i)))), Signed(\mathcal{P})$$

.

The prover signs a transaction θ from output γ which creates δ. However, the prover leaves θ unfinalized, without witnesses for the input data until the it decides on the input.

The output δ is constructed first.

The script for δ should be

$And(Or(After(\epsilon), Signed(\mathcal{V})), Signed(\mathcal{P}))$

Where ϵ is a timeout, which enables the prover to use the output if the verifier is withholding.

6 Program Construction

While arbitrary programs may be used, with arbitrary input and output values,

To maximize the usefulness of this protocol, the input to the circuit should contain two transactions:

- transaction which spends δ and ζ, using a signature from the verifier and prover with the nSequence on δ set to ψ.
- a partial transaction identical to the above, which is signed by the prover and not the verifier, with nSequence on δ set to 0.

These can be compressed to the diffs (just an additional signature with a different nSequence).

With these inputs, the spend of γ reveals the transactions that can be used with ζ.

If any output bit is incorrect, then the Verifier has at least ψ blocks before the Prover can use δ to complete the transaction, or to challenge the execution. If ψ blocks elapses, then the Prover can use the transaction themselves, or any transaction they prefer (TODO: not sufficiently bound, but still secure).

If the transactions are not valid according to the circuit, then a failure gate and outcome shall be triggered enabling the verifier to inflict punishment.

7 Optimal Garbling

BitGC: Garbled Circuits with 1 Bit per Gate has shown much more efficient garbling where marginal gates cost only 1-bit per gate. These constructions could make Delrag practical for implementation, but further study is needed.[3]

[3]Embarrassing confession: The author of this paper stopped working on it to understand this publication, but didn't find the time to review, so decided to publish this work in draft form.