TariScript: Bringing dynamic scripting to Mimblewimble

Cayle Sharrock and Schalk van Heerden

Tari Labs
{cj,sw}@tari.com

Abstract. Mimblewimble is a cryptocurrency protocol with good privacy and scalability properties. A trade-off of Mimblewimble is the requirement that transactions are interactive between sender and receiver. TariScript is presented as an extension to Mimblewimble that adds scripting capabilities to the protocol. We describe the theoretical basis for TariScript and present the modifications required to make it secure. The trade-offs and use cases for TariScript are briefly covered.

Keywords: Blockchain · Mimblewimble · Scripting · TariScript

1 Introduction

This paper introduces TariScript, a dynamic scripting extension for Mimblewimble. Scripting unlocks many applications, including support for unilateral payments, covenants, atomic swaps, and side-chain pegs. We briefly review how vanilla Mimblewimble handles transactions to illustrate why, for example, unilateral payments are impossible. We touch on what other projects are doing to overcome these limitations. We then detail the theoretical framework for TariScript and conclude with a description of a concrete implementation of TariScript, its use-cases and limitations.

1.1 Mimblewimble

Mimblewimble is a scalable, confidential cryptocurrency protocol developed by the anonymous developer, Tom Jedusor [1].

Confidentiality comes from the use of Pedersen Commitments [2] to blind values, rather than publishing transaction values in the clear, as in Bitcoin [3], for example.

The key scalability advantage that Mimblewimble gains over Bitcoin is that it is only necessary to know the emission schedule, the current unspent transaction output (UTXO) set, and some housekeeping data in order to verify that the accounting is correct. In particular, data associated with spent transaction outputs can be discarded without compromising the security guarantees of the protocol [1].

The key to the protocol is the homomorphic balance over the expected supply of coins and the set of outputs in circulation. The details are given in [1] and an accessible explanation is available at [4], but we provide a simplified summary of the key points here for convenience.

Given a blinding factor k_i and value v_i using generators from a suitable elliptic curve G and H respectively, we denote the Pedersen commitment C_i as the combination, $C_i \equiv v_i \cdot H + k_i \cdot G$.

The emission of new coins follows a pre-determined schedule such that s_b are the number of new coins minted in block b. For a chain containing M unspent outputs after N blocks, the following balance must hold

$$\sum_{i}^{M} C_i = \sum_{b}^{N} s_b + \Delta_N \tag{1}$$

where $\Delta_b = \delta_b \cdot G$ is termed the total accumulated public excess after block b. In general, the sum of all the blinding factors, k_i must equal the total accumulated excess, i.e.,

$$\delta_b = \sum_{i} k_{ib} \; \forall i \in \text{UTXO set after block } b \tag{2}$$

There is a similar balance for every Mimblewimble transaction:

$$\sum_{j \in \text{outputs}} C_j - \sum_{i \in \text{inputs}} C_i \equiv \text{fee} \cdot H + \Delta \tag{3}$$

where $\Delta = \delta \cdot G$ and the excess is again the sum of the blinding factors, taking note of a sign change between outputs and inputs, i.e.,

$$\delta = \sum_{j \in \text{outputs}} k_j - \sum_{i \in \text{inputs}} k_i$$

The public excess for the transaction is collectively calculated by the parties in the transaction. A signature committing to the public excess, signed by each party is stored in the blockchain data. A running total of the total accumulated public excess is maintained by validators to verify the balance in Eq. (1) after each block.

When miners validate a block, they lump every transaction into effectively a single transaction and calculate the block-equivalent of Eq. (3):

$$\sum_{j \in \text{outputs}} C_j - \sum_{i \in \text{inputs}} C_i \equiv \sum_{m \in \text{txs}} (\text{fee}_m \cdot H + \Delta_m) \tag{4}$$

1.2 Tari

Tari [5][6] is a Mimblewimble-based proof-of-work blockchain. Tari aspires to provide a scalable, private, digital assets network and smart contract platform. To achieve this, the Tari protocol requires several features unsupported by Mimblewimble, including unilateral (i.e. non-interactive) payments, covenants and side chain peg-in transactions [7].

A review of the prior art finds that some, not all, of these features may be possible to implement using "scriptless scripts" [8]. In particular, unilateral payments are not possible with scriptless scripts since they cannot avoid the interactive requirement of Mimblewimble transactions.

Unilateral payments were employed in LiteCoin extension blocks [9][10].

Beam [11], adds support for general smart contract applications via a virtual machine extension. Neither approach directly augments the core protocol with native scripting. We believe that Tari is unique in this approach.

In addition, we seek to devise a solution that unlocks all of Tari's desired use-cases with a single, generalised approach, hence TariScript.

2 TariScript

TariScript is a protocol extension that adds dynamic scripting capabilities to Mimblewimble. Some implementation details are omitted in the interests of brevity but a full description is available in the Tari specifications [12].

We begin with the observation that the balance equations (1) and (4) are necessary but not sufficient to secure Mimblewimble. Additional rules, including the inclusion of a range proof [13], proving that the value transferred is non-negative, and the excess signature, reduces the set of all possible transaction expressions to the subset of transactions considered secure under the Mimblewimble protocol.

TariScript adds another, strictly non-expansive, constraint to the spending rules of UTXOs. This means that TariScript only reduces the subset of valid transactions; never increases it. Therefore the security guarantees of Mimblewimble are not affected. While this argument does not constitute a security proof of TariScript, it is a strong intuitive argument that the approach is valid, with some caveats that we will cover shortly. A rigorous security proof for TariScript is an avenue for future work.

An outline of the TariScript specification is as follows:

1. Every UTXO must carry a script, α.
2. The script can accept arbitrary input parameters, θ, provided by the spender of the UTXO. The script is executed to produce a result, r, i.e., $\alpha(\theta) \rightarrow r$.
3. The result r must be singular (i.e. not an array) *and* must be a valid public key under G. If this is the case,
4. we assign the result, r to a new variable introduced in TariScript, the *script public key*, K_s,.
5. The UTXO may be spent if, other requirements notwithstanding, the spender demonstrates knowledge of *both* the commitment blinding factor k (this is the vanilla Mimblewimble requirement), and the script private key, k_s, such that $K_s = k_s \cdot G$. The latter is demonstrated by signing the script and its input with k_s.

There is an implicit caveat that the script and its metadata be non-malleable (no parties can modify it once the sender has broadcast the transaction) and immutable (it cannot be removed once it is in the blockchain).

To address malleability, the script, α, is signed and the signature is included with the UTXO. Additionally, a signature signing the script and the input parameters are provided when spending the UTXO.

To address immutability, we first review a feature of vanilla Mimblewimble called cut-through [1]:

If Alice sends coins to Bob, and Bob sends the same coins to Charlie in the same block, miners, at their discretion, may omit Bob's UTXO entirely, and Eq. (4) will still hold because Bob's commitments simply cancel out. Note that, only the commitment is cut-through. Since kernels are never cut through, and fees are specified in the kernel, the fees paid by Bob are still tracked and the overall accounting still balances.

Cut-through poses a problem for TariScript since Alice is required to provide script with UTXO B. If B is cut-through, the script is lost and we violate the immutability constraint of TariScript. Therefore a new mechanism, the script offset, γ, is introduced to explicitly prevent cut-through in Tari blocks.

2.1 Preventing cut-through with the script offset

First, the sender chooses a new secret scalar, k_o, from the curve which we call the sender-offset private key. The sender calculates the corresponding sender-offset public key, $K_o = k_o \cdot G$ and includes this key with the UTXO metadata.

As per the TariScript rules, the recipient must know the script private key, k_s, corresponding to the script public key, $K_s = k_s \cdot G$, which results from the script execution, $\alpha(\theta) \rightarrow r = K_s$. In other words, to spend an output, a spender must provide input, θ, to the script such that it resolves to some K_s for which spender knows the private key.

The recipient proves knowledge of k_s, by signing the script and input data with k_s when spending the output.

Then, the script offset, γ, is calculated by summing all script private keys for every input in the transaction, and subtracting the sum of all sender-offset private keys in the transaction. That is,

$$\gamma = \sum_{i \in \text{inputs}} k_{si} - \sum_{j \in \text{outputs}} k_{oj} \tag{5}$$

The script offset is broadcast along with the usual Mimblewimble transaction data. Any third party can verify that the script offset is correct by ensuring that

$$\gamma \cdot G \equiv \sum_{i \in \text{inputs}} K_{si} - \sum_{j \in \text{outputs}} K_{oj} \tag{6}$$

When miners construct blocks, they do a similar aggregation for script offsets that they do for transaction excess:

$$\gamma_{\text{total}} = \sum_{m \in \text{txns}} \gamma_{\text{m}} \tag{7}$$

It is a simple exercise to prove that Eqs. (5) and (6) apply for a transaction block if γ is replaced by γ_{total}.

The block script offset, γ_{total}, is stored in the block header.

It naturally follows that the script offset γ means that no third party can change or remove any input or output from a transaction or the block, since doing so will invalidate the script offset balance equations, (6) or its block-level equivalent. By extension, the script offset also prevents cut-through. In the scenario above, if a miner cut out B from the transactions between Alice, Bob, and Charlie, the script offset calculation would differ by $k_{sB} - k_{oB}$ and the validation would fail.

To address malleability concerns, both the script and sender-offset public key must be signed by the sender-offset private key. This signature is included with the UTXO. In addition, every input contains a valid script signature, which signs the script, the input data and the script public key with the script private key.

3 Implications and trade-offs

3.1 Script lock key generation

It may appear the burden for wallets has tripled since each UTXO owner has to remember three private keys: the spend key, k_i, the sender offset key k_o and the script key k_s for every transaction. In practice, this is not the case.

Spend keys are typically deterministically derived from a single seed phrase, for example, using hierarchical deterministic (HD) wallets [14].

On closer inspection of (6), the sender-offset private key does not actually need to be stored at all. Once the script offset, γ and script signature are calculated and broadcast, the offset is never required again, and can be discarded.

Script key management is application-dependent. For unilateral payments, the script key is a static key associated with the recipient's node or wallet, similar to an Ethereum address, and so only a single private key is required. For default payments, as shown in the examples, the recipient is free to provide any value they like, which can be discarded once the transaction is finalised. In other applications, the script key may derived from a suitable path in an HD wallet, as per the spend keys.

3.2 Blockchain size

The most obvious drawback to TariScript is the effect it has on blockchain size. UTXOs are substantially larger, with the addition of the script, metadata signature, script signature, and a public key to every output. The increase depends on the script and its serialisation, but typically, UTXOs are 13% - 40% larger than vanilla Mimblewimble. These can eventually be pruned but will nevertheless increase storage and bandwidth requirements. TariScript inputs are two to four times larger than vanilla Mimblewimble inputs. The latter consist of a commitment and output features. In TariScript, each input includes a script, input data, the script signature, and an extra public key. In addition, every block header contains an extra field, the total script offset, that cannot be pruned away.

In terms of performance, TariScript introduces two additional signature verification operations and an additional balance requiring a single curve operation. Script evaluation, $\alpha(\theta) \rightarrow r$, will depend on the implementation. Tari's implementation, as discussed in the next section, is very efficient with strict upper bounds on script size and complexity. In practice, we find that the Tari network is limited by block space and network constraints during busy periods rather than transaction processing speed [15].

3.3 Chain analysis

Another potential drawback of TariScript is the additional information that is handed to entities wishing to perform chain analysis. Having scripts attached to outputs will often clearly mark the purpose of that UTXO. Users may wish to re-spend outputs into standard, default UTXOs in a mixing transaction to disassociate TariScript funds from a particular script.

3.4 Horizon attacks

The Mimblewimble protocol allows outputs, and consequently, the output scripts to be discarded (pruned) once they are spent.

In practice, nodes maintain a cache of full blocks before pruning outputs. The depth of the cache is termed the pruning horizon. The horizon plays a key role in simplifying node synchronisation and handling short chain forks that inevitably arise in proof-of-work chains. Nodes are able to verify that scripts are respected as long as chain re-organisations are not deeper than the pruning horizon.

In particular, when a new node joins the network, it will not be able to know whether the scripts attached to spent transactions older than the pruning horizon were faithfully executed to reach the current chain head.

The usual Mimblewimble guarantees remain, including the overall coin balance, but there is now an avenue of attack for a malicious party: Force a chain re-organisation chain beyond the pruning horizon, and alter the script of a spent output for which the attacker knows the spend key; thus enabling spending of the output. This applies in the specific case of unilateral payments, where the attacker knows the spend key, but not the script key.

This is termed a "horizon attack". There are three ways to mitigate or prevent it:

1. After receiving a unilateral payment, the receiver can spend the output to herself with a standard interactive payment. This prevents the horizon attack completely. Wallets can be programmed to do this automatically and periodically, at the cost of an additional on-chain transaction, batching output spends to reduce on-chain costs.

2. Make the pruning horizon very deep, several weeks' worth, say, to the point that the cost of re-organising the chain for the horizon attack becomes uneconomical.
3. Run at least one node in archival node, meaning that UTXOs are never pruned. Only a single honest archival node across the entire network is required to be able to eventually bring all other nodes back into the correct consensus.

All three mitigations strategies can be run independently and in concert to reduce the risk of a horizon attack to one of theoretical concern.

4 Examples

4.1 Standard Mimblewimble transactions

The simplest script that maintains the current Mimblewimble payment functionality is the identity script,

$$\alpha(\theta) \to \theta \tag{8}$$

Here the spender can provide an arbitrary public key as the script input and it will be interpreted as the script public key, K_s. In practice, a spender would use a nonce, $r = k_s$, with $K_s = k_s \cdot G$ and the UTXO commitment's blinding factor, k, which they alone know, to spend the output in a transaction.

4.2 Unilateral payments

The strategy for leveraging TariScript for unilateral payments is as follows: Assume Alice wants to pay Bob at some static "address". Mimblewimble does not have addresses, *per sè*, but any public key, $K_{sb} \equiv k_{sb} \cdot G$ that Bob makes publicly available will suffice.

The problem can be reduced to one in which Alice is able to publish the transaction unilaterally, such that Bob can independently identify and claim the output without any communication from Alice, and such that Alice cannot spend the output herself.

The solution centers on Alice using Bob's public key as the script public key, K_s. Alice then combines her sender-offset key, k_{ob}, and Bob's public key, to derive a shared secret using a Diffie-Hellman key exchange [16]:

$$k_b = \mathrm{H}\big(k_{ob} \cdot K_{sb}\big) = \mathrm{H}\big(k_{sb} \cdot K_{ob}\big) \tag{9}$$

where H is a suitable hash function that produces a valid scalar under G. Alice also encrypts the value of the commitment with the shared secret and stores it at any convenient location in the transaction metadata.

This concludes part one of the problem.

To prevent Alice being able to spend the transaction, she provides a script, $\alpha(\cdot) \to K_{sb}$. That is, the script returns the script public key when a null input is provided. Since Alice does not know k_{sb}, she cannot sign the script when trying to spend the input. However, Bob can and thus part two of the problem is solved.

Once broadcast, any node can verify that the transaction is complete, verify the signature on the script, and verify the script offset.

Bob can scan all transactions looking for a script that matches $\alpha(\cdot) \rightarrow K_{sb}$. If so, he recovers the spend key using his private key (also the script private key in this case) and the sender-offset public key as per the third term in Eq. (9).

4.3 TariScript script - a concrete α implementation

Tari [5] uses a simple Forth-like stack-based language [17], similar to Bitcoin script in its implementation of α. We use the term TariScript to refer to both the protocol modifications and the set of opcodes and execution rules that define the script language.

Scripts contain opcodes representing commands that are executed sequentially. The commands operate on a single data stack that initially contains the input data. The set of commands include mathematical and cryptographic operations, stack manipulation and conditional logic [18].

After the script completes, it is successful if and only if it has not aborted and there is exactly a single element remaining on the data stack. The script fails if the stack is empty, contains more than one element, or aborts early.

To prevent denial-of-service or resource exhaustion attacks, both the script and stack are limited in size. Any stack overflow results in script failure. In addition, there are no opcodes for loops or timing functions, guaranteeing that every script will terminate.

Example script - time-locked spending conditions Interesting and complex transactions can be constructed dynamically, and without needing to make any further changes to the Mimblewimble protocol.

As an example, Alice wants to send some Tari to Bob. But, if he doesn't spend it within a certain time frame (up till block 4000), then she wants to be able to spend it back to herself.

This type of transaction is impossible in vanilla Mimblewimble and, in fact, is also outside the reach of scriptless scripts, which have no concept of blockchain context, like the block height.

However, the TariScript script in figure 1 achieves the desired result.

```
Dup PushPubkey(P_b) CheckHeight(4000) GeZero IFTHEN
PushPubkey(P_a) OrVerify(2) Drop
ELSE EqualVerify ENDIF
```

Fig. 1. An example time-locked contract in TariScript. This script serialises to 84 bytes, of which 64 bytes are taken up by the two public key representations.

The spender (Alice or Bob) provides their public key, some `P_x`, as input to the script. The first opcode, `DUP`, duplicates the top element of the stack, leaving two copies of the public key. Then, `PushPubkey(P_b)`, pushes Bob's public key, `P_b`, onto the stack.

Subsequently, `CheckHeight(4000)` pushes the difference between the current block and block 4000 to the stack. `GeZero` replaces the top stack element with 1 if the value was positive, or 0 if it was negative.

`IFTHEN` pops an element of the stack and executes the commands up until the `ELSE` opcode if the value is equal to one, and the commands between `ELSE` and `ENDIF` otherwise.

Let's assume the chain is at block 3990. All the commands up to `EqualVerify` are skipped. `EqualVerify` pops two items off the stack, which currently contains `P_b,P_x,P_x` and does nothing if they are equal, or aborts if they are not. Thus, the script will only continue if `P_b == P_x`.

The script is now complete, leaving the single value of `P_x` on the stack, which must be Bob's public key. This key is used as the script public key, K_s, as per the TariScript rules. Bob will have signed the script and its input with k_s as part of the valid transaction.

Alternatively, if the block height is 4000 or above, then the expression in line 2 from Figure 1 is executed. First, `PushPubkey(P_a)` pushes Alice's public key onto the stack, leaving a data stack `P_a,P_b,P_x,P_x`.

Then `OrVerify(n)` pops n items off the stack. If the new top element is equal to *any* of the popped elements, the script continues, otherwise the script aborts. In this example then, both Alice and Bob's public keys are popped and the script will continue if, and only if, `P_x` matches one of them.

Assuming this is the case, the `Drop` opcode drops the superfluous key, leaving `P_x`, which must be one of Alice or Bob's public keys. As before the spender will also have provided a suitable signature using their private key.

5 Conclusion

TariScript provides a novel way of extending Mimblewimble with dynamic scripting abilities, while retaining its scaling and confidential properties. This is offset by larger outputs and blocks. The scripts are also subject to horizon attacks, which must be mitigated with a long pruning horizon or prevented by output sweeping and running at least one archival node. The benefits significantly outweigh the drawbacks, since this single extension enables multiple features needed to give Mimblewimble generalised smart contract capabilities, including unilateral payments, covenants and side-chain pegs.

Acknowledgements The authors are indebted to David Burkett for reviewing several iterations of TariScript and for initially suggesting the basis for the script offset. We would also like to thank Hansie Odendaal, Michael Berry, Stanley Bondi and Phillip Robertson for critically reviewing initial drafts of TariScript, suggesting improvements and helping with the implementation in Tari.

References

1. Jedusor, T.: Mimblewimble. https://docs.beam.mw/Mimblewimble.pdf. (2016).
2. Pedersen, T.P.: Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. Advances in Cryptology — CRYPTO '91. 129–140. https://doi.org/10.1007/3-540-46766-1_9.
3. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf. (2008).
4. Sharrock, C.: Mimblewimble transactions explained, https://tlu.tarilabs.com/protocols/mimblewimble-transactions-explained, (2020).
5. Tari: The protocol for digital assets. https://tari.com. Last accessed 6 April 2023.
6. The Tari RFCs https://rfc.tari.com/. Last accessed 6 April 2023.
7. Back, A., Corallo M., Dashjr L., Friedenbach M., Maxwell G., Miller A., Poelstra A., Timón J., Wuille, P.: Enabling Blockchain Innovations with Pegged Sidechains. https://blockstream.com/sidechains.pdf. (2014).
8. Gibson, A.: Schnorrless Scriptless Scripts. https://reyify.com/blog/schnorrless-scriptless-scripts. (2020).
9. Burkett, D.: LIP004. https://github.com/DavidBurkett/lips/blob/master/lip-0004.mediawiki. Last accessed 29 Mar 2023
10. Burkett, D., Lee, C., Yang, A: LIP003. https://github.com/litecoin-project/lips/blob/master/lip-0003.mediawiki. Last accessed 3 Apr 2023
11. One side payments. https://github.com/BeamMW/beam/wiki/One-side-payments. (2019). Last accessed 4 April 2023.
12. Sharrock, C., Berry, B., van Heerden, S., Odendaal, H.,: RFC-201: TariScript. https://rfc.tari.com/RFC-0201_TariScript.html. Last accessed 6 April 2023.
13. Bünz, B., Bootle J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short Proofs for Confidential Transactions and More.Cryptology ePrint Archive. https://eprint.iacr.org/2017/1066.pdf. (2017).
14. Wuille, P.: Hierarchical Deterministic Wallets https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki. Last accessed 17 May 2023.
15. Odendaal, H.: Stress test of 2021/12/21 (Console to Mobile wallets) https://github.com/tari-project/tari-data-analysis/blob/master/reports/stress_tests/20211214-make-it-rain/Stress%20test%20of%2020211214%20-%20analysis.md. Last accessed 17 May 2023.
16. Merkle, R.: Secure Communications Over Insecure Channels. Communications of the ACM. **21**(4), 294–299 (April 1978)
17. Moore, C.: Programming a problem-oriented language http://forth.org/POL.pdf. (1970).
18. Sharrock, C.: RFC-202: TariScript Opcodes. https://rfc.tari.com/RFC-0202_TariScriptOpcodes.html. Last accessed 6 April 2023.