Σ-State, an Alternative to Bitcoin Script

December 11, 2017

Abstract

We report on design and implementation of Σ-State, a more powerful, protocol-friendly and at the same more secure language than the Script language used in Bitcoin. We provide some examples of scripts in the language.

1 Introduction

Since very early days, Bitcoin [?] is not just about simple money transfers: it has a scripting language named Bitcoin Script. The language is a primitive stack-based language without loops. A program in this language is protecting a transaction output. To spend the output, a spending transaction must provide a script in the same language, such as concatenation of both scripts is to be interpreted in *true* value on top of the stack. Combined with a maximum size of a 10 kilobytes, the absence of loops provides a guarantee of strictly finite validation time. However, some denial-of-service attacks were found against script validation time. kushti notes : links

Smart contracts platform Ethereum is allowing arbitrary program to be run on top of blockchain. kushti notes : continue

Our paper is introducing a new language, which is more powerful than Bitcoin Script, and more friendly to cryptographic protocols and applications. It is also providing better guarantees for validation to be finished within target timeout. kushti notes : continue

The idea behind the language is that a subset of zero-knowledge protocols known as Σ-protocols (sigma protocols) could be combined via $\wedge$ and $\vee$ connectives forming complex statements like "prove me a knowledge of discrete logarithm of (publicly known) x_1 or knowledge of Pedersen commitment x_2 preimage". We augment such statements with predicates over state of blockchain system during script validation (which happens when a transaction tries to spend an output protected by the script). To avoid inefficient processing and impossibility for a light client to validate a transaction, the state is minimal but nevertheless it is richer than in Bitcoin. Like in Bitcoin, we use current height of the blockchain, but also a spending transaction with its inputs and outputs, and output which contains script is also exposed to it, along with height when it was created. Unlike Bitcoin, we allow outputs to contain more fields than amount and protecting script, in form of additional registers. In addition to $\wedge$ and $\vee$ connectives, we have different operations over typed arguments. More, we filter out wrong expressions, like "2 + 2 ¿ true", thanks to a type system used.

To tackle the problem of denial-of-service attack carrying by crafting scripts which are too costly to validate (for example, if it is needed for more time to validate a script than average delay between blocks on commodity hardware, network could be obviously attacked, with nodes stuck in processing and increased number of forks as result).

1.1 Related Work

kushti notes : Simplicity kushti notes : Plutus kushti notes : Typecoin?

1.2 Organization of the Paper

2 Language Design

2.1 Notation

We use $dlog(x)$ to denote a statement "prove a knowledge of such w that $x = g^w$". Proving is to be done in zero knowledge (with no presenting the secret w). Statement $dlog(x_1) \wedge dlog(x_2)$ means "prove a knowledge of both w_1 and w_2, such as $x_1 = g^{w_1}, x_2 = g^{w_2}$", similarly, $dlog(x_1) \vee dlog(x_2)$ is about a proof knowledge of whether w_1 or w_2.

2.2 Model

We assume that a *prover* and a verifier have a shared context. We assume that this context is about current state of the blockchain (such as height of a best block h), spending transaction and an output to spend.

An output to spend is protected by a logical statement, which is a mix of predicates over known context. As a simplest example, consider the following statement:

$$dlog(x_1) \vee ((h > 5) \wedge dlog(x_2)) \tag{1}$$

which is to be read as "proof of knowledge of a secret with public image x_1 is always enough, also, if height of a block containing spending transaction is more than 5, knowledge of a secret with public image x_2 is also enough".

Both prover and verifier are first reducing the statement by substituting shared context variables. Four outcomes of the reduction are possible: *true*, *false*, failure to reduce (if statement is invalid) which is equivalent to *false*, or statement which contains only cryptographic statements. For the example, if $h = 10$, the reduced statement is $dlog(x_1) \vee dlog(x_2)$. In this case, the prover is generating a proof, and verifier is checking validity of the proof, accepting or rejecting it. We use cryptographic statements which are provable via *sigma protocols* (kushti notes : links). These protocols are kind efficient Zero-Knowledge Proof-of-Knowledge protocols which are composable via AND, OR and k-out-of-N conjectures, also any sigma protocol has a standard way to be converted into a signature by using the Fiat-Shamir transformation (kushti notes : link).

In addition to secret information, which knowledge is to be proven in zero-knowledge, we allow prover to enhance context with custom variables. For example, for the statement:

$$dlog(x) \wedge (blake2b256(c) = C)$$

where C is some constant [1], and $blake2b256$ is operation which calculates hash value for function Blake2b256 (kushti notes : link). The prover then needs to prove knowledge of discrete logarithm of x and also to present value c such as Blake2b256 on c evaluates to C. Even if verifier does not know c before being presented a proof, we can not count c as secret, as it could be replayed by an eavesdropper.

[1]we avoid provide a value for the constant due to column size limit

2.3 Types

2.4 Language Details

kushti notes : better subsection name

We now provide details on building blocks of the language.

By parsing a statement, interpreter first builds a tree from a formula. For the example 1 the tree would be as following:

kushti notes : draw the tree.

A tree is then to be transformed in following steps (we provide here steps for the verifier, the prover is repeating some steps twice as described in details in Section 2.5):

kushti notes : all transformations are bottom-up

1. context-subst Substitute custom variables. From context extension formed by the Prover, during this step custom variable names in the stamenent are to be replaced by their respective values.
2. ss-subst Complex context substitutions. This step is about substitutions which can increase size of a statement. kushti notes : $self.script, tx.outbytes, tx.output, tx.has\_output$
3. var-subst Trivial context substitutions. Substitutions to be made in this step do not increase statement size. kushti notes : $height, self.height, self.amount$
4. operations Operations
5. relations Relations
6. conjectures Conjectures

The process of the transformations could be aborted by an exception, if expression to reduce is not valid (for example, a wrong type of argument), or if it is found during the transformations that the tree becomes too complex to finish its processing on time (details are provided in the Section 4). The result of the transformation could be a single tree node containing a boolean value (for example, *true* for the statement $2 + 2 > 3$) or a tree which contains only cryptographic sub-statements and logical conjectures (for example, $dlog(x_1) \vee dlog(x_2)$). In the latter case, the prover is proving statement validity, and the verifier checks the reduced statement against the proof.

2.5 Proving

kushti notes : Raw text from Leo's email, rewrite

Prover steps:

1. bottom-up: mark every node real or simulated, according to the following rule. DLogNode – if you know the secret, then real, else simulated. $\vee$: if at least one child real, then real; else simulated. $\wedge$: if at least one child simulated, then simulated; else real. Note that all descendants of a simulated node will be later simulated, even if they were marked as real. This is what the next step will do.

Root should end up real according to this rule – else you won't be able to carry out the proof in the end.

2. top-down: mark every child of a simulated node "simulated." If two or more more children of a real $\vee$ are real, mark all but one simulated.

3. top-down: compute a challenge for every simulated child of every $\vee$ and $\wedge$, according to the following rules. If $\vee$, then every simulated child gets a fresh random challenge. If $\wedge$ (which means $\wedge$ itself is simulated, and all its children are), then every child gets the same challenge as the $\wedge$.

4. bottom-up: For every simulated leaf, simulate a response and a commitment (i.e., second and first prover message) according to the Schnorr simulator. For every real leaf, compute the commitment (i.e., first prover message) according to the Schnorr protocol. For every $\vee/\wedge$ node, let the commitment be the union (as a set) of commitments below it.

5. Compute the Schnorr challenge as the hash of the commitment of the root (plus other inputs – probably the tree being proven and the message).

6. top-down: compute the challenge for every real child of every real $\vee$ and $\wedge$, as follows. If $\vee$, then the challenge for the one real child of $\vee$ is equal to the XOR of the challenge of $\vee$ and the challenges for all the simulated children of $\vee$. If $\wedge$, then the challenge for every real child of $\wedge$ is equal to the the challenge of the $\wedge$. Note that simulated $\wedge$ and $\vee$ have only simulated descendants, so no need to recurse down from them.

2.6 Verifying

Verifier steps: 1. Place received challenges "e" and responses "z" into every leaf.

2. Bottom-up: compute commitments at every leaf according to $a = g^z/h^e$. At every $\vee$ and $\wedge$ node, compute the commitment as the union of the children's commitments. At every $\vee$ node, compute the challenge as the XOR of the children's challenges. At every $\wedge$ node, verify that the children's challenges are all equal. (Note that there is an opportunity for small savings here, because we don't need to send all the challenges for a $\wedge$ – but let's save that optimization for later).

3. Check that the root challenge is equal to the hash of the root commitment and other inputs.

2.7 Threshold Signatures

3 Examples

In this section we provide some examples of useful contracts. We focus on examples which are impossible or harder to realize in Bitcoin.

3.1 Crowdfunding

We provide very simple solution to crowdfunding here. In the example, a crowdfunding project associated with public key x_P is considered successful if it is possible to collect for the projects unspent outputs with total value not less than *to_raise* before height *timeout*. The project backer creates an output protected by following statement:

$$(height \geq timeout \wedge dlog(x_B)) \vee (height < timeout \wedge dlog(x_P) \wedge tx.has\_output(amount \geq to\_raise \wedge script = dlog(x_P$$

Then the project can collect biggest outputs with total value not less than *to_raise* with a single transaction (it is possible to collect up to 22,000 outputs in Bitcoin, which is enough even for a big crowdfunding campaign). For remaining small outputs over *to_raise*, it is possible to construct follow-up transactions.

Please note the extension which is not a part of Bitcoin: we use the condition on a spending transaction, namely, we require the transaction to have an output with value not less than required and also with particular statement protecting the output.

3.2 Money With Scheduled Maintenance Payments

kushti notes : description

$user\_statement \vee (height \geq (self.height + period) \wedge tx.has\_output(amount \geq (self.amount - cost) \wedge script = self.sc$

We highlight impossibility of such a statement in the Bitcoin Script: similarly to the statement in the previous section 3.1, we use a condition on a spending transaction. Another feature missed in the Bitcoin Script is that we also use an output to spend in the execution context. In particular, in the example above we require a spending transaction to have an output which has the same statement as an output it spends.

3.3 Ring Signature

Linear-sized ring signature is very straightforward in the language. Assume a ring consists of m public keys $x_1, \dots, x_m$. If we want an output to be spent by a ring signature associated with the ring, the output is to be protected by the following statement:

$$dlog(x_1) \vee \cdots \vee dlog(x_m)$$

Please note that proving of the statement is to be done in zero-knowledge, so it is not possible to know which key signed, the only fact to conclude is that some key from the ring signed output spending.

3.4 Complex Signature Schemes

We can build more complex signature schemes than possible in Bitcoin. One particular example was provided in the previous Section 3.3. Another example is a scheme where at least one out of (Alice, Bob), and at least one out of (Charles, Diana) are needed to sign, and it is not to be known who signed an output spending. The corresponding statement involving public keys x_A, x_B, x_C, x_D of Alice, Bob, Charles and Diana respectively would be following:

$$(dlog(x_A) \vee dlog(x_B)) \wedge (dlog(x_C) \vee dlog(x_D))$$

3.5 Simple Tumbler

Privacy is a tough problem for cryptocurrency users. Bitcoin is a pseudonymous cryptocurrency, so no real identities attached to a transaction. However, it is possible to reconstruct transactional graph for all the transactions ever entered into the Bitcoin blockchain, and often restore real identities by using auxiliary databases. kushti notes : link To improve privacy, tumblers are being used. A tumbler is a scheme which us getting some money transfers as inputs, produces output money transfers, and has a property of unlinkability: it is not possible to draw a link from an input to an output. Thus a tumbler user is hiding herself amongst a ring of users sending inputs to the scheme. Privacy then depends on a ring size. For maximum privacy, there exists a cryptocurrency ZCash with inbuilt tumbler based on zkSnarks kushti notes : links, where a user is hiding among all the users in the system. However, ZCash requires trusted setup, and transaction validation is relatively slow (10 ms). In other cryptocurrencies users are usually using external services varying in security and efficiency.

Here we are describing simple tumbler service. It is very efficient and requires no trusted dealer. Its disadvantage is that from observing blockchain transactions it is possible to conclude that users are using a tumbler.

Assume Alice with public key x_A and Bob with public key x_B want to relocate funds to keys y_1 and y_2 respectively, with a property than external observer looking into blockchain is not capable to know beyond the fact that money flows from x_A to whether y_1 or y_2 (and the same for x_B).

First, Alice and Bob communicate off-chain to construct collectively outputs of the final transaction (payments to y_1 and y_2). Each of them then is calculating a hash value from outputs bytes h (we assume that Blake2b-256 hash function is used). Then each of them is creating an output to spend (possibly, in a dedicated transaction) with such a condition for Alice and Bob, respectively:

$blake2b256(tx.outbytes) = h \vee dlog(x_A)$

$blake2b256(tx.outbytes) = h \vee dlog(x_B)$

Then it is possible to make a refund at any moment of time (right condition in the $\vee$ conjectures), and before a refund any of the them can construct a transaction which is spending the outputs, and it is impossible to construct an alternative transaction, for which hash of the output bytes is h, but bytes are different (if chosen hash function is collision-resistant).

4 Safety Guarantees

We need to be sure that an adversary can not produce such a statement for which the Verifier spends more time than it is safe to spend. kushti notes : links to verifier dilemma, orphan rates etc

4.1 Cost Model

4.2 Denial-of-Service Attacks

5 Extensibility

It is hard to predict which functions would be useful for blockchain applications.

6 Implementation and Evaluation

7 Further Work

8 Conclusion

We show in this paper...

A Cost Table

B Roadmap

C Outputs vs Accounts