Is it Possible to Verify if a Transaction is Spendable?

Marcelo Arenas[1,2,3]*, *Thomas Reisenegger*[1,3], *Juan Reutter*[1,2,3] and *Domagoj Vrgoč*[1,2,3]

[1]*Department of Computer Science, Universidad Catolica de Chile, Santiago, Chile,* [2]*Institute for Mathematical and Computational Engineering, Universidad Catolica de Chile, Santiago, Chile,* [3]*IMFD Chile, Santiago, Chile*

With the popularity of Bitcoin, there is a growing need to understand the functionality, security, and performance of various mechanisms that comprise it. In this paper, we analyze Bitcoin's scripting language, Script, that is one of the main building blocks of Bitcoin transactions. We formally define the semantics of Script, and study the problem of determining whether a user-defined script is well-formed; that is, whether it can be unlocked, or whether it contains errors that would prevent this from happening.

Keywords: bitcoin, script, static analysis, unlockability, script transaction

Edited by:
Cosimo Laneve,
University of Bologna, Italy

Reviewed by:
Roberto Zunino,
University of Trento, Italy
Francesco Tiezzi,
University of Camerino, Italy

****Correspondence:***
Marcelo Arenas
marenas@ing.puc.cl

Specialty section:
This article was submitted to Smart Contracts, a section of the journal Frontiers in Blockchain

Received: *03 September 2021*
Accepted: *15 October 2021*
Published: *14 December 2021*

Citation:
Arenas M, Reisenegger T, Reutter J and Vrgoč D (2021) Is it Possible to Verify if a Transaction is Spendable? Front. Blockchain 4:770503. doi: 10.3389/fbloc.2021.770503

1 INTRODUCTION

Bitcoin (Nakamoto, 2008; Bonneau et al., 2015; Narayanan et al., 2016; Antonopoulos, 2017) is a decentralized cryptocurrency protocol proposed in 2008 by a person or a group of people under the pseudonym Satoshi Nakamoto. As a currency, Bitcoin allows for transactions between users, and can be used for instance as a way of transferring money between individuals in a secure way, and without depending on any bank or centralized institution. But there are several other advantages of using Bitcoin to transfer currency. The subject of this article is a feature called "smart contracts", which, in Bitcoin, work by specifying certain requirements that must be satisfied by transactions before this money can be spent[1]. These contracts are issued using *Script*, a language specifically designed for this task, and that is integrated into the Bitcoin protocol.

The Bitcoin protocol and its Script language permit the design of different forms of smart contracts, and currently we have a variety of pre-designed contracts, and several formal models to understand the correctness of contracts, their semantics or their power [see e.g. (Bartoletti and Zunino, 2019)]. However, there are still lower-level complexity questions that remain unanswered about Script. In this paper, we focus on the complexity of processing scripts, and, more importantly, of verifying whether a smart contract is *valid*, in the sense that the requirements posed by the contract are actually possible to satisfy. In order to dig deeper on Bitcoin's smart contracts, we start by pointing out some of the differences that exist between a common bank transaction and a Bitcoin transaction.

First, there is no concept of account in the Bitcoin protocol. Assume that person A wants to transfer X amount of money to person B. A does not have an account with a balance that determines how much money he/she can transfer. Instead, A must point to one or more transactions of which he/she is the recipient, and whose sum must be at least X. Clearly, the system must address the problem of determining which transaction outputs have been spent and which have not. In the case of Bitcoin, instead of inspecting the whole ledger to determine whether a certain transaction output

[1]This notion should not be confused with the more general notion of smart contract in Ethereum (Buterin et al., 2014; Dannen, 2017), which has been developed based on a Turing-complete language (Atzei et al., 2017; Antonopoulos and Wood, 2018).

has been spent, the nodes in the network keep a record of all of the unspent transaction outputs (UTXOs).

The second main difference between bank and Bitcoin transactions is that the Bitcoin protocol was designed to allow for more complex spending requirements. In other words, instead of just indicating a recipient for a transaction, the sender states certain requirements that need to be met by the recipient in order to spend the transferred money. For example, one could wish to forbid the money from being spent before a certain date, or to require multiple people to agree to spend the money. The tool that is used to establish these requirements is Script, which is a non-Turing-complete scripting language designed specifically for this purpose (O'Connor, 2017; Klomp and Bracciali, 2018; Jansen et al., 2019).

Script was designed to disallow infinite loops from being created, so that the nodes in the network could not be tricked into executing a never-ending program. However, the requirements that can be represented through it can be complex, and this is why these requirements can be understood as smart contracts.

In practice, the protocol for spending requirements associates each transaction output with a locking script, which corresponds to a sequence of Script operators. Afterwards, when creating a new transaction, in addition to pointing to an unspent transaction output, the sender must provide an unlocking script that fulfills the requirements established through the locking script associated with such an output. Specifically, to determine if the unlocking script is valid, the nodes that receive these transactions append the locking script to the unlocking script, execute the resulting construction and determine whether the execution is successful. An execution is considered successful if it does not raise any errors and results in a structure that represents the Boolean value *true*.

Script provides enough freedom to easily create a locking script for which there does not exist any valid unlocking script. This can be done on purpose, and there is even a specific operator *OP_RETURN* that automatically flags the locking script as invalid. In practice, this is used to store information in the blockchain, so that there is a verifiable proof that said information was available to the sender on a certain date. However, locking scripts that cannot be unlocked can also be created by mistake.

This causes problems at the individual and collective level. On the one hand, a person simply loses money if he/she creates a transaction with a locking script that cannot be unlocked. In fact, there is no possible way of accessing funds that have been locked in this manner. On the other hand, these unspent transactions are accumulated in the pool of UTXOs, occupying memory and resources on all the nodes that have received it. Given that these outputs cannot be spent, the resources used to manage them cannot be freed.

Our goal is to understand the complexity of determining whether the output of a transaction is spendable or not, by looking at how its associated locking script is constructed. As a first necessary contribution for tackling this goal, we propose a simple and direct formalization of a fragment of Script, which provides a suitable setting to define and study the aforementioned unlockability problem. We use our formalization to prove that there is no efficient algorithm for detecting unspendable transaction outputs in the considered fragment of Script (unless Ptime = NP), which immediately implies that no such an algorithm can exists for the entire language. Interestingly, we also use our formalization to provide a mathematical proof for the folklore fact that processing a script is in Ptime.

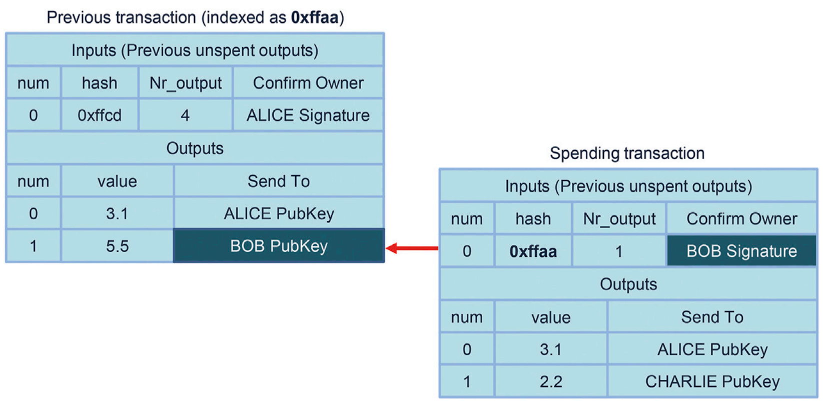
Our formalization of Script is similar to the one presented in (Klomp and Bracciali, 2018); in particular, they are both based on a notion of configuration, or state, that is updated when a Script operator is executed. A state is defined in (Klomp and Bracciali, 2018) as a single main stack together with some extra components like pointers to the head and bottom elements of the stack, and the semantics of Script is defined by a set of structural operation semantics rules. On the other hand, the notion of configuration in our formalization consists of the main and alternate stacks used in Script, and a control stack needed to define *if* statements. We diverted from the definition in (Klomp and Bracciali, 2018) to have a more appropriate formalization to study the unlockability problem, which also includes the alternate stack of Script. A comprehensive description of different formalizations and extensions of Script can be found in (Bartoletti and Zunino, 2018). These works have focused on proposing executable semantics of Script, and some extensions of it, and on enabling the formal verification of some properties of protocols defined in this language (Andrychowicz et al., 2014; O'Connor, 2017; Atzei et al., 2018b; Atzei et al., 2018a; Bartoletti and Zunino, 2019; Singh et al., 2020). In this sense, our definition of Script follows a different direction, guided by the need to study the unlockability problem, which, to the best of our knowledge, has not been considered in previous works.

2 HOW SCRIPT WORKS

Transactions are at the core of Bitcoin. Simply put, they specify which coins are spent and to whom they are transferred. On a technological level, each Bitcoin transaction can have multiple inputs, each of which is an output of a previous transaction. Conceptually, for a transaction to be accepted, each input that is used requires a digital signature that corresponds to the public key specified by the transaction where this input was generated[2]. We depict this dependence graphically in **Figure 1**. Besides, the list of all transactions (grouped into blocks) is kept by a peer-to-peer network "running" Bitcoin, so that we are able to check if the transaction inputs have already been spent. The only transactions that differ from this template are the coinbase transactions in which new "coins" are minted, and that have no inputs. These appear once per block, and only specify who can spend the newly created "coins".

In reality, the process of signing a transaction input is more complicated and depends on Bitcoin's scripting language, Script. More precisely, each transaction output specifies a part of a script

[2]As we explain below, one does not necessarily provide a digital signature. This example serves for illustrative purposes only.



Previous transaction (indexed as **0xffaa**)

Inputs (Previous unspent outputs)			
num	hash	Nr_output	Confirm Owner
0	0xffcd	4	ALICE Signature

Outputs		
num	value	Send To
0	3.1	ALICE PubKey
1	5.5	BOB PubKey

Spending transaction

Inputs (Previous unspent outputs)			
num	hash	Nr_output	Confirm Owner
0	**0xffaa**	1	BOB Signature

Outputs		
num	value	Send To
0	3.1	ALICE PubKey
1	2.2	CHARLIE PubKey

FIGURE 1 | The input to one transaction is the output of a previous transaction. Here Bob confirms with his digital signature that he is the owner of the private key corresponding to the public key used when specifying the recipient of the funds in the previous transaction. Transactions reference each other *via* their hash (i.e. 0xffaa in this case).

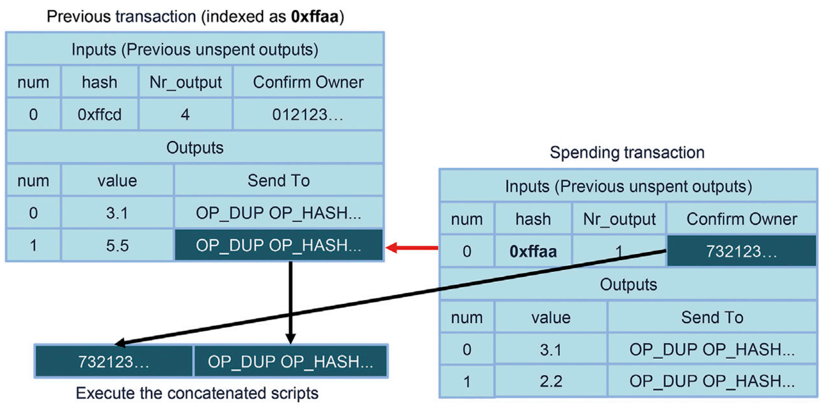
written in this language, called the locking script. In order to spend this output, the transaction using it as an input must provide another sequence of Script commands, called the unlocking script, such that the script obtained by concatenating the two executes correctly. Given that stack-based languages operate "in-reverse", the two scripts are also concatenated in this order, namely, the locking script is appended to the unlocking script spending it. We depict this process graphically in **Figure 2**.

When Script was conceived, the process of executing the combination of both scripts was done by literally concatenating them together, and then executing the resulting script. However, for safety concerns this procedure has been modified, so that the execution of the concatenation is performed by first executing the unlocking script while checking that it was properly constructed, and then executing the locking script with the final state of the execution of the unlocking script as its initial state (Github, 2010). This distinction is irrelevant in the analysis of the most commonly used locking scripts. However, it will become important in the later sections of this document, when laying out proofs about the inner workings of Script.

Script (Bitcoin Wiki, 2021) is a simple stack-based language which allows to push elements to a stack, and manipulate its content using basic arithmetic, logical operations, if-else statements, and cryptographic primitives such as hashing and signature verification. Script is designed to be loop-free and is, therefore, not Turing-complete (O'Connor, 2017), which allows it to be more secure, and to be implemented efficiently. In spite of this, Script still allows to express an array of complicated conditions, giving rise to what is known as "smart contracts", which are nothing more than non trivial Script programs that specify how an output of a previous transaction can be unlocked. In what follows, we briefly recap the main commands of Script, and explain the problems we study in this setting.

Script evaluation relies on a stack in order to store some elements, perform simple operations on them, and later compare them for equality. Instructions of Script can be grouped as follows:

- Data (256 bit numbers), which are pushed onto the stack when encountered.
- Stack operations (push, pop, . . .).



Previous transaction (indexed as **0xffaa**)

Inputs (Previous unspent outputs)			
num	hash	Nr_output	Confirm Owner
0	0xffcd	4	012123...

Outputs		
num	value	Send To
0	3.1	OP_DUP OP_HASH...
1	5.5	OP_DUP OP_HASH...

Spending transaction

Inputs (Previous unspent outputs)			
num	hash	Nr_output	Confirm Owner
0	**0xffaa**	1	732123...

Outputs		
num	value	Send To
0	3.1	OP_DUP OP_HASH...
1	2.2	OP_DUP OP_HASH...

732123...	OP_DUP OP_HASH...

FIGURE 2 | Interaction between the locking and unlocking scripts.

- Logical operations (and, or, . . .).
- Arithmetical operations on numbers.
- Cryptographic primitives (hashing and signature verification).

We show how basic Script commands work, by illustrating how a basic transaction to transfer funds from one address to another works. This is called pay to public-key hash (or P2PKH for short) script, and is one of the simplest scripts that can be expressed.[3] As stated previously, each input to a transaction has an associated locking script. In the case of P2PKH, this locking script is as follows:

OP_DUP OP_HASH160 pubKeyData OP_EQUALVERIFY OP_CHECKSIG.

To unlock this output, we need to provide a set of Script commands, which, when executed prior to executing the locking script, result in a stack with a nonzero element at the top. A correct unlocking script in this case would be

signature pubKeyData.

Intuitively, the unlocking script provides us the signature signature and the public key pubKeyData corresponding to this signature, and then the locking script checks its validity. Namely, the locking script duplicates the top item on the stack (*via* OP_DUP), hashes this element (with a combination of ripeMD160 and SHA-256 hash functions), pushes an item onto the stack, pushes the public key data onto the stack, checks that the provided public key and the one specified in the script match, and finally verifies the signature.

This example already shows how locking scripts can specify complex conditions. While it is easy to construct the unlocking script for the locking script above, provided we have the required private key needed to produce the signature, this is not necessarily always the case. For instance, the locking script.

OP_DUP OP_ADD 7 OP_EQUALVERIFY

can never be unlocked since it is asking for an integer number n such that $2n = 7$. This can of course be very problematic if funds are locked behind such a locking script. A good Bitcoin wallet should try to prohibit such transactions, or at least try to warn the user that his/her output will become unspendable due to the locking script condition. This is known as the unlockability problem, and it is the main subject of study of this paper. More precisely, we provide a formalization of a fragment of the language Script in **Section 3**. Then we use this formalization in **Section 4** to provide a definition of the unlockability problem, and to prove that this problem is NP-hard. Finally, a discussion of the consequences of this intractability result are given in **Section 5**.

3 FORMALIZING SCRIPT

In this section, we develop a formalization for Script that allows us to study the computational complexity of some problems related to the evaluation or unlocking of scripts. Besides, this formalization enables us to fix the notation used throughout the paper. Given that Script is a stack-based language, we begin with a formal definition of the stacks that are used by this language. We then focus on the operators of Script, defining their semantics in terms of stack operations.

3.1 The Stacks in Script

For an arbitrary nonempty set M, we denote the concatenation of two elements $A, B \in M$ as $A \cdot B$, and naturally extend this notion to any finite number of elements. By $M*$ we denote all finite concatenations of elements of M, including the empty string ε, and with M^+ we denote $M*$ without ε. A stack over M is any element $A_0 \cdot A_1 \cdots A_k \in M*$. Intuitively, this string over M represents a stack containing A_0 as the top element, A_1 as the element below the top one, etc. Notice that we allow the empty stack, which is denoted by the empty string ε.

Script has two stacks at its disposal: the main stack, denoted by φ_M, and an alternate stack, denoted by φ_A, that can be accessed by a few of the operators. Hence, the stacks of Script shall be denoted as the pair (φ_M, φ_A). To manipulate these stacks, we use functions *top* and *tail*, defined as follows: *top*: $M^+ \rightarrow M$ is used to return the top of the stack, that is $top(A_0 \cdot A_1 \cdot \ldots \cdot A_k) = A_0$, while *tail*: $M^+ \rightarrow M*$ is used to return the stack below the first element, that is, $tail(A_0 \cdot A_1 \cdot \ldots \cdot A_k) = A_1 \cdot \ldots \cdot A_k$. Notice that the result of *tail* can be the empty stack ε.

3.2 Script Operators

For simplicity, we assume that data items in Script come from the set $\mathbb{Z}$.[4] This is a natural generalization when studying the complexity of the unlockability problem for Script.

Script has a precisely defined set of allowed operations (Bitcoin Wiki, 2021), which can be thought of as transforming the two stacks, or giving an error that terminates the execution. We denote the set of Script operators with O. Formally, every Script command f, apart from those used for flow control (see **Section 3.2.3**), can be understood as a function which takes the main and the alternate stack as its inputs, and transforms them in some way, or produces an error (denoted by $\square$):

$$f\colon\ (\mathbb{Z}^* \times \mathbb{Z}^*) \cup \{\square\} \rightarrow (\mathbb{Z}^* \times \mathbb{Z}^*) \cup \{\square\}. \tag{1}$$

Thus, scripts–as functions–can be composed, which naturally allows us to define the semantics of a sequence of operators. In particular, to handle errors, we impose the restriction that all of Script operators return an error when the input is an error itself, that is, $f(\square) = \square$.

With this notation at hand, we define how each operator $f \in O$ works. We start by introducing in **Section 3.2.1** a group of basic operators, and defining how a sequence of them is executed. Then we describe in **Section 3.2.2** how the operators associated with cryptographic primitives work. Finally, we introduce in **Section**

[3]We use this for simplicity. Pay to script hash is by far the currently most used type of script, often encapsulating P2PKH.

[4]In other words, we assume that each binary string encodes an integer. Notice that in Bitcoin the integers are bounded by size, however, for complexity theoretic analysis it is natural to lift this restriction, since considering bounded size inputs makes all of the results trivial.

3.2.3 the *flow control* operators and the control stack, which determine when an operator should or should not be executed. A summary of the operators used in this paper, without including the control flow operators, is given in **Table 1**. Readers familiar with the Script syntax as given in (Bitcoin Wiki, 2021) may note that a small number of the operators are not included in this table. For space reasons we have left out several operators that are similar to or can be simulated by applying instead a constant number of other operators. This includes, as explained bellow, merging all push operators into a single family of operators, arithmetic operators OP_1ADD and OP_1SUB for adding or subtracting one from the top of the stack, OP_NEGATE to flip the sign of the top of the stack, OP_ABS for the absolute value, binary operations OP_NOT, OP_0NOTEQUAL, OP_BOOLAND and OP_BOOLOR, number comparison operators OP_NUMEQUAL, OP_NUMEQUALVERIFY, OP_NUMNOTEQUAL, which are not useful under the assumption that elements in the stack are numbers, and comparison operators OP_LESSTHAN, OP_GREATERTHAN, OP_LESSTHANOREQUAL, OP_GREATERTHANOREQUAL, OP_MIN, OP_MAX and OP_WITHIN. Reserved words are not included because they immediately make transactions invalid, and similarly for pseudo words OP_PUBKEYHASH, OP_PUBKEY and OP_INVALIDOPCODE. An operator-by-operator check allows one to verify that none of these operators alter the results presented in this paper, and in particular the PTIME upper bound shown later on in the paper continues to hold. Furthermore, we have also left out locktime-related operators OP_CHECKLOCKTIMEVERIFY and OP_CHECKSEQUENCEVERIFY, because they require checking the nLockTime field of transactions and this is not part of our model. Finally, we will comment on the cryptographic operators on **Section 3.2.2**; we have also left out some of them but once again the complexity analysis does not change.

3.2.1 Basic Operators in Script

The most basic operation in Script is pushing data onto the (main) stack, which is achieved using a multitude of different operators [see e.g. the section on "Constants" in Bitcoin Wiki (2021)]. In order to simplify this process, we combine all of these methods of pushing data through the *OP_PUSH$_C$* operator, which pushes the value C onto the main stack. In terms of our generic description of Script commands (1), the semantics of this operation is defined as follows:

$$\text{OP\_PUSH}_C(\varphi_M, \varphi_A) = (C \cdot \varphi_M, \varphi_A).$$

That is, if the operator receives as input a pair of valid stacks φ_M and φ_A, then it puts C on top of φ_M. Moreover, as already mentioned, we assume that *OP_PUSH$_C$*(□) = □.

Notice that for each value $C \in \mathbb{Z}$, we include an operator *OP_PUSH$_C$*. We designed the language in this way to be able to define the semantics of a sequence of operators as their composition as functions. In fact, if we had included a single operator *OP_PUSH* with input (C, φ_M, φ_A), then this property would no longer hold; in particular, the output (φ_M, φ_A) of an operator cannot be used as the input of *OP_PUSH*.

Similarly, to pop the top of the stack, we can use *OP_DROP*, and to duplicate the top element of the stack, *OP_DUP*. Both of these operators require that the main stack φ_M contains at least one element (i.e. $|\varphi_M| \geq 1$), otherwise they return an error. In the case of a nonempty stack, their behavior is defined as:

$$\begin{aligned}\text{OP\_DROP}(\varphi_M, \varphi_A) &= (\text{tail}(\varphi_M), \varphi_A)\\ \text{OP\_DUP}(\varphi_M, \varphi_A) &= (\text{top}(\varphi_M) \cdot \varphi_M, \varphi_A)\end{aligned}$$

The alternate stack in Bitcoin can be accessed in a very limited number of ways: we can only move the top element from the main stack onto it by means of the operator *OP_TOALTSTACK*, and move the top element of the alternate stack onto the main stack by means of the operator *OP_FROMALTSTACK*. Formally,

$$\begin{aligned}\text{OP\_TOALTSTACK}(\varphi_M, \varphi_A) &= (\text{tail}(\varphi_M), \text{top}(\varphi_M) \cdot \varphi_A) && \text{if } |\varphi_M| \geq 1\\ \text{OP\_FROMALTSTACK}(\varphi_M, \varphi_A) &= (\text{top}(\varphi_A) \cdot \varphi_M, \text{tail}(\varphi_A)) && \text{if } |\varphi_A| \geq 1\end{aligned}$$

In **Table 1**, we provide the list of remaining basic operators and their semantics (except for the last three rows of this table that include the operators defined in the following section).

As Script operators are understood as functions, the semantics of a script $f_1 \cdot f_2 \cdot \ldots \cdot f_n$ consisting of a sequence of operators is defined as the composition of these functions. Moreover, a script is *executed successfully* over a stack φ if upon executing all of its commands with φ as the initial main stack, we are left with a nonempty main stack containing a nonzero element at the top. Formally, a script $f_1 \cdot f_2 \cdot \ldots \cdot f_n$ is executed successfully over a stack φ if $(f_n \circ \ldots \circ f_2 \circ f_1)(\varphi, \varepsilon) = (\varphi_M, \varphi_A)$ with $\varphi_M \neq \varepsilon$ and $top(\varphi_M) \neq 0$. It is important to notice that the possibility of starting with a nonempty main stack is included because of the way in which the unlocking and the locking script are executed in succession, which does not exactly match the execution of the concatenation of both scripts. Formally, when we have a locking script l, and an unlocking script u, we require that: 1) $u(\varepsilon, \varepsilon) = (\varphi_M^u, \varphi_A)$ (with no errors thrown in between); and 2) $l(\varphi_M^u, \varepsilon)$ executes successfully.

Example 3.1. Consider the script

$$\text{OP\_PUSH}_5 \cdot \text{OP\_PUSH}_{-3} \cdot \text{OP\_ADD}$$

We execute this script starting with empty main and alternate stacks. We first push number 5 onto the main stack, and then push −3 at the top of the main stack. The last operator is OP_ADD, which according to the semantics defined in **Table 1** generates a main stack containing only the number 2 = − 3 + 5. Hence, this script is executed successfully, since upon its completion, we have a nonempty main stack with a nonzero top element. ■

3.2.2 Operators for Executing Cryptographic Primitives

An important part of Script resides in the execution of cryptographic primitives, since in most of the popular locking scripts these functions are used to verify the identity of the recipient of a transaction. While there are several cryptographic operators in Script, we only consider the most prevalent of them: *OP_HASH*160, which hashes an

TABLE 1 | Semantics of Script commands. We assume that $\varphi_M = A_0 \cdot A_1 \cdots A_k$ whenever $\lvert\varphi_M\rvert > 0$. The condition column states the requirement that needs to be met for each operator not to return an error. Formally, if the condition for operator *f* is not met by (φ_M, φ_A), then $f(\varphi_M, \varphi_A) = \square$. The function *hash* corresponds to using SHA-256 and RIPEMD-160 hashing algorithms in succession. The function *chksig* corresponds to the verification algorithm of the ECDSA protocol for the string comprised of the transaction information, the first input as the public key and the second input as the signature. Computing the transaction information is a non-trivial process in Bitcoin. Since the main focus in this paper is to study the properties of Script itself, we do not model this process in our formalization.

Operator	Condition	Semantics
OP_PUSH$_C$	None	$OP\_PUSH_C(\varphi_M, \varphi_A) = (C \cdot \varphi_M, \varphi_A)$
OP_DROP	$\lvert\varphi_M\rvert \geq 1$	$OP\_DROP(\varphi_M, \varphi_A) = (tail(\varphi_M), \varphi_A)$
OP_DUP	$\lvert\varphi_M\rvert \geq 1$	$OP\_DUP(\varphi_M, \varphi_A) = (top(\varphi_M) \cdot \varphi_M, \varphi_A)$
OP_TOALTSTACK	$\lvert\varphi_M\rvert \geq 1$	$OP\_TOALTSTACK(\varphi_M, \varphi_A) = (tail(\varphi_M), top(\varphi_M) \cdot \varphi_A)$
OP_FROMALTSTACK	$\lvert\varphi_A\rvert \geq 1$	$OP\_FROMALTSTACK(\varphi_M, \varphi_A) = (top(\varphi_A) \cdot \varphi_M, tail(\varphi_A))$
OP_VERIFY	$\lvert\varphi_M\rvert \geq 1 \wedge top(\varphi_M) \neq 0$	$OP\_VERIFY(\varphi_M, \varphi_A) = (tail(\varphi_M), \varphi_A)$
OP_IFDUP	$\lvert\varphi_M\rvert \geq 1$	$OP\_IFDUP(\varphi_M, \varphi_A) = \begin{cases} (\varphi_M, \varphi_A) & \text{if} \quad top(\varphi_M) = 0 \\ (top(\varphi_M) \cdot \varphi_M, \varphi_A) & \text{if} \quad top(\varphi_M) \neq 0 \end{cases}$
OP_NIP	$\lvert\varphi_M\rvert \geq 2$	$OP\_NIP(\varphi_M, \varphi_A) = (A_0 \cdot A_2 \cdot \ldots \cdot A_k, \varphi_A)$
OP_OVER	$\lvert\varphi_M\rvert \geq 2$	$OP\_OVER(\varphi_M, \varphi_A) = (A_1 \cdot \varphi_M, \varphi_A)$
OP_ROT	$\lvert\varphi_M\rvert \geq 3$	$OP\_ROT(\varphi_M, \varphi_A) = (A_2 \cdot A_0 \cdot A_1 \cdot A_3 \cdot \ldots \cdot A_k, \varphi_A)$
OP_SWAP	$\lvert\varphi_M\rvert \geq 2$	$OP\_SWAP(\varphi_M, \varphi_A) = (A_1 \cdot A_0 \cdot A_2 \cdot \ldots \cdot A_k, \varphi_A)$
OP_TUCK	$\lvert\varphi_M\rvert \geq 2$	$OP\_TUCK(\varphi_M, \varphi_A) = (A_0 \cdot A_1 \cdot A_0 \cdot A_2 \cdot \ldots \cdot A_k, \varphi_A)$
OP_2DROP	$\lvert\varphi_M\rvert \geq 2$	$OP\_2DROP(\varphi_M, \varphi_A) = (tail(tail(\varphi_M)), \varphi_A)$
OP_2DUP	$\lvert\varphi_M\rvert \geq 2$	$OP\_2DUP(\varphi_M, \varphi_A) = (A_0 \cdot A_1 \cdot \varphi_M, \varphi_A)$
OP_3DUP	$\lvert\varphi_M\rvert \geq 3$	$OP\_2DUP(\varphi_M, \varphi_A) = (A_0 \cdot A_1 \cdot A_2 \cdot \varphi_M, \varphi_A)$
OP_2OVER	$\lvert\varphi_M\rvert \geq 4$	$OP\_2OVER(\varphi_M, \varphi_A) = (A_2 \cdot A_3 \cdot \varphi_M, \varphi_A)$
OP_2ROT	$\lvert\varphi_M\rvert \geq 6$	$OP\_2ROT(\varphi_M, \varphi_A) = (A_4 \cdot A_5 \cdot A_0 \cdot A_1 \cdot A_2 \cdot A_3 \cdot A_6 \cdot \ldots \cdot A_k, \varphi_A)$
OP_2SWAP	$\lvert\varphi_M\rvert \geq 4$	$OP\_2SWAP(\varphi_M, \varphi_A) = (A_2 \cdot A_3 \cdot A_0 \cdot A_1 \cdot A_4 \cdot \ldots \cdot A_k, \varphi_A)$
OP_ADD	$\lvert\varphi_M\rvert \geq 2$	$OP\_ADD(\varphi_M, \varphi_A) = ((A_0 + A_1) \cdot A_2 \cdots A_k, \varphi_A)$
OP_SUB	$\lvert\varphi_M\rvert \geq 2$	$OP\_SUB(\varphi_M, \varphi_A) = ((A_1 - A_0) \cdot A_2 \cdots A_k, \varphi_A)$
OP_EQUAL	$\lvert\varphi_M\rvert \geq 2$	$OP\_EQUAL(\varphi_M, \varphi_A) = \begin{cases} (1 \cdot A_2 \cdot \ldots \cdot A_k, \varphi_A) & \text{if} \quad A_0 = A_1 \\ (0 \cdot A_2 \cdot \ldots \cdot A_k, \varphi_A) & \text{if} \quad A_0 \neq A_1 \end{cases}$
OP_EQUALVERIFY	$\lvert\varphi_M\rvert \geq 2 \wedge A_0 = A_1$	$OP\_EQUALVERIFY(\varphi_M, \varphi_A) = (A_2 \cdot \ldots \cdot A_k, \varphi_A)$
OP_PICK	$\lvert\varphi_M\rvert \geq 1 \wedge A_0 \geq 0 \wedge \lvert\varphi_M\rvert \geq A_0 + 2$	$OP\_PICK(\varphi_M, \varphi_A) = (A_{A_0+1} \cdot A_1 \cdot \ldots \cdot A_k, \varphi_A)$
OP_ROLL	$\lvert\varphi_M\rvert \geq 1 \wedge A_0 \geq 0 \wedge \lvert\varphi_M\rvert \geq A_0 + 2$	$OP\_ROLL(\varphi_M, \varphi_A) = (A_{A_0+1} \cdot A_1 \cdot \ldots \cdot A_{A_0} \cdot A_{A_0+2} \cdot \ldots \cdot A_k, \varphi_A)$
OP_DEPTH	None	$OP\_DEPTH(\varphi_M, \varphi_A) = (\lvert\varphi_M\rvert \cdot \varphi_M, \varphi_A)$
*OP_HASH*160	$\lvert\varphi_M\rvert \geq 1$	$OP\_HASH160(\varphi_M, \varphi_A) = (hash(A_0) \cdot tail(\varphi_M), \varphi_A)$
OP_CHECKSIG	$\lvert\varphi_M\rvert \geq 2$	$OP\_CHECKSIG(\varphi_M, \varphi_A) = (chksig(A_0, A_1) \cdot A_2 \cdot \ldots \cdot A_k, \varphi_A)$
OP_CHECKSIGVERIFY	$\lvert\varphi_M\rvert \geq 2 \wedge chksig(A_0, A_1) = 1$	$OP\_CHECKSIGVERIFY(\varphi_M, \varphi_A) = (A_2 \cdot \ldots \cdot A_k, \varphi_A)$

input, and *OP_CHECKSIG* and *OP_CHECKSIGVERIFY*, which are used to check a digital signature. The analysis for all the other cryptographic primitives is identical to these cases. Let us first describe the primitives *hash* and *chksig* underlying these operators.

The operator hash: $\mathbb{Z} \rightarrow \mathbb{Z}$ is a function whose value is the result of hashing the input by using SHA-256 and then RIPEMD-160. Moreover, *chksig* is defined as follows. In a digital signature protocol, the signature verification function receives as input a public key, a string and a signature. Then such a function determines whether the signature was obtained by executing the signing function over the string and the private key corresponding to the public key. However, the signature verification operators in Script only receive as input a public key and a signature. This is because the purpose of these operators is just to determine if the recipient has access to a certain private key. Therefore, the string that is signed is a predetermined construction that is obtained by executing certain transformations over a combination of the transaction's inputs, outputs and locking scripts. Thus, given that for the purposes of each script the document that is signed is a constant, we will disregard this element in our analysis, and we define chksig: $\mathbb{Z} \times \mathbb{Z} \rightarrow \{0, 1\}$ as a function that takes only two inputs: a string representing a public key and a string representing a digital signature. The value of $chksig(n_1, n_2)$ is defined as 1 if n_2 is a valid signature for the document constructed from the transaction (as described previously) and the public key n_1, and the value of $chksig(n_1, n_2)$ is 0 otherwise. The digital signature protocol that is used to generate and verify signatures is ECDSA with the secp256k1 elliptic curve (Bitcoin Wiki, 2021).

Finally, we provide the formal definitions of the hashing and signature checking operators. For the hashing operator, the main stack φ_M is required to contain at least one element (i.e. $\lvert\varphi_M\rvert \geq 1$), whereas both signature checking operators require the main stack to have at least two elements (i.e. $\lvert\varphi_M\rvert \geq 2$). If these conditions are not satisfied, then these operators return an error $\square$. In the definition, we assume that $\varphi_M = A_0 \cdot A_1 \cdot \ldots \cdot A_k$:

$$\begin{aligned}
\mathrm{OP\_HASH160}(\varphi_M, \varphi_A) &= (\mathrm{hash}(A_0) \cdot \mathrm{tail}(\varphi_M), \varphi_A) \\
\mathrm{OP\_CHECKSIG}(\varphi_M, \varphi_A) &= (\mathrm{chksig}(A_0, A_1) \cdot A_2 \cdot \ldots \cdot A_k, \varphi_A) \\
\mathrm{OP\_CHECKSIGVERIFY}(\varphi_M, \varphi_A) &= \begin{cases} (A_2 \cdot \ldots \cdot A_k, \varphi_A) & \text{if chksig}(A_0, A_1) = 1 \\ \square & \text{if chksig}(A_0, A_1) = 0 \end{cases}
\end{aligned}$$

3.2.3 Operators for Flow Control

The final piece we need to add are the flow control operators of the form if-then-else. While conceptually simple, formalizing this concept needs an extra piece of notation, since in a block of the form

if $<$ `somecommands` $>$ else $<$ `someothercommands` $>$ end_if, we need to determine the correct block of commands to be executed while reading the script from left to right. We achieve this by including an extra stack, called the *control stack*, which is denoted by φ_I. Intuitively, the control stack allows us to decide whether an operator is outside an if-then-else block, in which case it is executed as usual, or whether it belongs to some of the commands within this if-then-else block, in which case we need to make sure that only the operators from the appropriate block are being executed.

The control stack φ_I consist of zeros and ones exclusively, that is, $\varphi_I \in \{0,1\}^*$. A control stack φ_I is said to represent an *execution state* if $\varphi_I \in \{1\}^*$, which indicates that the command we are seeing has to be executed (in this case, this will be a command within the if-then-else block). Similarly, an empty control stack indicates that we are outside the if-then-else portion of the script, and should therefore execute the operator.

Working with an additional stack also requires to redefine the semantics of all other commands that we outlined in the previous sections, to allow us to work with them in case flow control operators are present in the script. We do this in the expected way: all previous operators are only executed when the control stack is in an execution state. That is, for every Script operator $f \in O$, **Eq. 1** should be replaced by the following:

$$f\colon\ (\mathbb{Z}^* \times \mathbb{Z}^* \times \{0,1\}^*) \cup \{\square\} \rightarrow (\mathbb{Z}^* \times \mathbb{Z}^* \times \{0,1\}^*) \cup \{\square\}. \quad (2)$$

Hence, each operator takes as input three stacks: the main stack, the alternate stack and the control stack. The semantics of commands from **Table 1** is then redefined so that there is a third input, φ_I, which is also the third output (φ_I is not changed by the operators in **Table 1**). Besides, the condition column in **Table 1** is modified to include the fact that φ_I represents an execution state (that is, $\varphi_I \in \{1\}*$). In particular, for each operator f in **Table 1**, if φ_I is not an execution state, then we have that $f(\varphi_M, \varphi_A, \varphi_I) = (\varphi_M, \varphi_A, \varphi_I)$; namely, the command is not executed. For example, consider again the operation *OP_PUSH*$_C$ with $C \in \mathbb{Z}$. Its semantics, taking now into consideration the control stack, is defined as follows:

$$\text{OP\_PUSH}_C(\varphi_M, \varphi_A, \varphi_I) = (C \cdot \varphi_M, \varphi_A, \varphi_I),$$

whenever φ_I is an execution state. When φ_I is not an execution state, the semantics of this operator is defined as:

$$\text{OP\_PUSH}_C(\varphi_M, \varphi_A, \varphi_I) = (\varphi_M, \varphi_A, \varphi_I).$$

The flow control operators *OP_IF*, *OP_ELSE*, *OP_ENDIF* are the only ones that can modify the control stack.

Next we explain how they interact with the main and alternate stacks, and also how they modify the control stack. In essence, these three commands come in tandem, and take the form:

$$\text{OP\_IF} < \texttt{commands1} > \text{OP\_ELSE} < \texttt{commands2} > \text{OP\_ENDIF}.$$

Both commands1 and commands2 are sequences of Script commands, which can again contain if-then-else blocks. The objective of the control stack is to signal whether commands1 or commands2 are to be executed, depending on whether the top value of the main stack upon reaching the *OP_IF* is true or false. This is achieved by pushing/popping the appropriate value to/from the control stack when either *OP_IF* or *OP_ELSE* is reached, as to signal which block of commands will be executed. Recall that only a control stack in an execution state allows for a command to be executed, so we will use this property accordingly.

Intuitively, when reaching an *OP_IF* statement, we will store the truth value of the top of the main stack onto the control stack. If this was true (or nonzero in our notation), we will push 1 onto the control stack, thus making it be in an execution state. Then, upon reaching its corresponding *OP_ELSE*, we will replace the value 1 at the top of the control stack with 0, making it not be in an execution state. This will allow us to skip all the commands until reaching the accompanying *OP_ENDIF*, which simply pops the top of the control stack. A similar process occurs when the top value of the main stack upon reaching *OP_IF* is false. Notice that if-then-else statements can be nested. However, in a syntactically correct script this is not an issue, as the control stack is populated and cleared as expected. Formally, the semantics of *OP_IF* is defined as follows:

$$\text{OP\_IF}(\varphi_M, \varphi_A, \varphi_I) = \begin{cases} (\text{tail}(\varphi_M), \varphi_A, 1 \cdot \varphi_I) & \text{if } |\varphi_M| \geq 1 \wedge \text{top}(\varphi_M) \neq 0 \wedge \varphi_I \in \{1\}^* \\ (\text{tail}(\varphi_M), \varphi_A, 0 \cdot \varphi_I) & \text{if } |\varphi_M| \geq 1 \wedge \text{top}(\varphi_M) = 0 \wedge \varphi_I \in \{1\}^* \\ (\varphi_M, \varphi_A, 0 \cdot \varphi_I) & \text{if } |\varphi_I| \geq 1 \wedge \varphi_I \notin \{1\}^* \end{cases}$$

Moreover, in any other case, *OP_IF*$(\varphi_M, \varphi_A, \varphi_I) = \square$. For example, an error is returned if φ_M is an empty stack, as there is no stack element to ascertain the truth value. Thus, the definition of OP_IF states that three outcomes are possible upon reaching this operator, under the appropriate conditions not to produce an error: 1) If the top element of the main stack is different from 0 and we are in an execution state, then 1 is pushed onto the control stack, in order to signal that the IF part of the if-then-else block is to be executed. Besides, the main stack is popped. 2) If the top of the main stack is 0, and we are in an execution state, we push 0 onto the control stack (i.e. we do not execute the commands in the IF block, but rather in the ELSE block), and the main stack is popped. 3) Finally, if we are not in an execution state, we push the value 0 to the control stack. The value of the element pushed to the control stack in this last case is not actually relevant because even if an OP_ELSE command where to invert it, the stack would remain in a state of no execution stemming from the existence of one or more 0 elements corresponding to lower if-then-else blocks. However, it is still necessary to track the existence of this new branch, in order to correctly close it once we reach its corresponding OP_ENDIF command.

On the other hand, the OP_ELSE operator simply has to signal whether the commands that follow it are to be executed or not, which is done by changing the top element of the control stack as follows:

$$\text{OP\_ELSE}(\varphi_M, \varphi_A, \varphi_I) = \begin{cases} (\varphi_M, \varphi_A, 1 \cdot \text{tail}(\varphi_I)) & \text{if } |\varphi_I| \geq 1 \wedge \text{top}(\varphi_I) = 0 \\ (\varphi_M, \varphi_A, 0 \cdot \text{tail}(\varphi_I)) & \text{if } |\varphi_I| \geq 1 \wedge \text{top}(\varphi_I) = 1 \end{cases}$$

Moreover, if φ_I is empty, then the operator *OP_ELSE* returns an error, that is, *OP_ELSE*$(\varphi_M, \varphi_A, \varphi_I) = \square$. Notice that OP_ELSE simply flips the top bit in the control stack to signal the transition between the "then" and the "else" blocks. As previously stated, in case the top element in the stack is a 0 because the stack was not in

PROBLEM	UNLOCKABILITY OF SCRIPT
INPUT	A locking script *l*
QUESTION	Is there an unlocking script *u*, such that *l* and *u*, when given as inputs to the EVALUATION OF SCRIPT problem result in a positive answer?

an execution state when reaching the corresponding OP_IF operator, OP_ELSE will change it to a 1. However, the stack will still not be in an execution state, because there will still exist one or more 0 elements that stem from lower if-then-else blocks. Finally, each if-then-else block is required to be correctly closed *via* the *OP_ENDIF* operator. To ensure this, we simply pop the top element of the control stack upon reaching this command:

$$\text{OP\_ENDIF}(\varphi_M, \varphi_A, \varphi_I) = (\varphi_M, \varphi_A, \text{tail}(\varphi_I))$$

Notice that as for the case of *OP_ELSE*, if φ_I is empty, then the operator *OP_ENDIF* returns the error symbol $\square$.

It is important to notice that in adding these flow control operators to Script, we introduce more nuance into the definition of a successful execution. More specifically, we now say that a script is *executed successfully* over a stack φ if upon executing all of its operators with φ as our initial main stack, we are left not only with a nonempty main stack which contains a nonzero element at the top, but also with an empty control stack. Formally, a script $f_1 \cdot f_2 \cdot \ldots \cdot f_n$ is executed successfully over a stack φ if $(f_n \circ \cdots \circ f_2 \circ f_1)(\varphi, \varepsilon, \varepsilon) = (\varphi_M, \varphi_A, \varphi_I)$ with $\varphi_M \neq \varepsilon$, $top(\varphi_M) \neq 0$ and $\varphi_I = \varepsilon$. Conceptually, this new condition requires flow control blocks to be properly structured in Script. In particular, a script that ends with a nonempty control stack has an unfinished if-then-else block, which indicates that it is not well constructed.

As we have explained previously, when executing a pair of an unlocking and a locking script, the process consists of executing the unlocking script over a trio of empty stacks, and then executing the locking script over the final main stack of the previous execution and a pair of empty stacks. However, if after the first execution we are left with a nonempty control stack signaling unfinished if-then-else blocks, then the locking script is simply given an error and the combined execution ends unsuccessfully (see next section for a formal definition of the unlockability of Script problem). Therefore, when executing a pair of an unlocking and a locking script, both executions have to contain properly structured if-then-else blocks.

Example 3.2. To illustrate how flow control operators work, consider the following script:

```
OP_PUSH$_0$
OP_IF
    OP_DUP
OP_ELSE
    OP_PUSH$_3$
    OP_IF
        OP_PUSH$_7$
    OP_ELSE
        OP_DUP
    OP_ENDIF
OP_ENDIF
```

Recall that a script consists of a concatenation of operators, but we have represented this vertically and indented to better illustrate how flow control blocks are nested. When executing this script, value 0 is pushed onto the main stack first (notice that at the beginning the control stack is empty, and we are thus in an execution state), so we have that:

$$(\varphi_M, \varphi_A, \varphi_I) = (0, \varepsilon, \varepsilon)$$

Following this, an OP_IF statement is encountered, and the control stack is updated accordingly. In this case, given that we have value 0 on top of the main stack, 0 is pushed onto the control stack, and the main stack is emptied:

$$(\varphi_M, \varphi_A, \varphi_I) = (\varepsilon, \varepsilon, 0)$$

Since we are not in an execution state, the OP_DUP command is ignored, and we continue with the OP_ELSE operator. Given that the top of the control stack is equal to 0, we replace this value with 1, signaling that the next block of commands is to be executed:

$$(\varphi_M, \varphi_A, \varphi_I) = (\varepsilon, \varepsilon, 1)$$

The operator OP_PUSH$_3$ is then executed, so the value 3 is pushed onto the main stack:

$$(\varphi_M, \varphi_A, \varphi_I) = (3, \varepsilon, 1)$$

Afterwards, another OP_IF operator is reached. Since we are in an execution state and value 3 is different from 0, value 3 is popped from the main stack, and 1 is pushed onto the control stack:

$$(\varphi_M, \varphi_A, \varphi_I) = (\varepsilon, \varepsilon, 1 \cdot 1)$$

This means that in the next step we push value 7 onto the main stack, when executing the operator OP_PUSH$_7$:

$$(\varphi_M, \varphi_A, \varphi_I) = (7, \varepsilon, 1 \cdot 1)$$

The next operator is OP_ELSE, which switches the value 1 on top of the control stack to 0, which in turn means that we are no longer in an execution state:

$$(\varphi_M, \varphi_A, \varphi_I) = (7, \varepsilon, 0 \cdot 1)$$

Thus, we need to ignore the following OP_DUP operator, and we need to continue with the OP_ENDIF command. Here the top of the control stack is popped:

$$(\varphi_M, \varphi_A, \varphi_I) = (7, \varepsilon, 1)$$

Finally, the last command OP_ENDIF is executed, leaving the control stack empty, and finishing with value 7 on the main stack:

$$(\varphi_M, \varphi_A, \varphi_I) = (7, \varepsilon, \varepsilon)$$

Thus, the script results in a successful execution. ■

Finally, we comment once again that some flow-control operators have been left out from our formalization. We do this for the sake of readability and because those operators can be simulated with a constant number of other operators. In particular, the operators we do not covered correspond to: OP_NOP, which does nothing at all, OP_NOTIF which simulates an OP_IF but when the value of the top of the stack is false, and OP_RETURN, which is used to mark transactions as invalid.

4 COMPLEXITY OF SCRIPT

In this section, we will focus on analyzing the computational cost of working with Script. To draw a complete picture, we start by formally defining in **Section 4.1** the evaluation and unlockability problems for this language. Then in **Section 4.2**, we provide a formal proof for the folklore result that evaluating a pair of unlocking and locking scripts can be done in polynomial time for the set of Script operators currently in use (Bitcoin Wiki, 2021). We also highlight that the original implementation of Script contained some operators that actually allowed for the construction of programs that run in exponential time in the length of the script, so the disabling of these is well justified. Moreover, we show in **Section 4.3** that the situation with the unlockability problem is completely different, as this problem is shown to be NP-hard. It is important to mention that this latter result is proved by combining some of the simplest operators in Script, and, in particular, without relaying on any of the cryptographic operators in the language. Hence, this result is a warning that the unlockability problem can become difficult even if some simple operators are used.

PROBLEM	EVALUATION OF SCRIPT
INPUT	A locking script l and an unlocking script u
QUESTION	Are the following executions successful (i) $u(\varepsilon, \varepsilon, \varepsilon) = (\varphi_M^u, \varphi_A^u, \varepsilon)$; and (ii) $l(\varphi_M^u, \varepsilon, \varepsilon)$?

4.1 The Evaluation and Unlockability Problems

The evaluation problem for Script is defined as follows:

As explained in **Section 2**, the unlocking script u, and the locking script l are executed separately in order to strengthen the security of Script. That is, we first run the unlocking script with a triple of empty stacks. Provided that this execution is successful, the content of the main stack at the end of this execution, denoted by φ_M^u, is transferred to the locking script, whose execution starts with an empty alternate stack and an empty control stack. In fact, as per the current specification (Bitcoin Wiki, 2021), and implementation (Github, 2010) of Script, the alternate stack content is erased when starting the execution of the locking script. Recall from **Section 3** the fact that a successful execution also requires the script to start and finish with an empty control stack, to validate that flow control commands are properly nested and completed within both the locking and the unlocking script. From now on, when the answer to the previous question is positive, then we say that the pair of scripts u and l executes successfully.

Moreover, the unlockability of Script problem is defined as follows:

4.2 On the Complexity of the Evaluation Problem

While the main objective of this paper is studying unlockability of Script, we will start by proving the folklore result saying that any script can be evaluated in polynomial time. We do this to show that our formalization of Script conforms with the intuitive understanding of the language. It is important to note that if we were to add some simple operators to Script that may seem unassuming, this property could cease to be true. In fact, previous versions of the language were able to produce scripts that could not be evaluated in polynomial time. To illustrate this notion we introduce the currently disabled OP_MUL operator. Let $\varphi_M = A_0 \cdot \ldots \cdot A_k, \varphi_A \in \mathbb{Z}^*$ and $\varphi_I \in \{1\}*$. We define the semantics of OP_MUL as follows. If $|\varphi_M| \geq 2$, then

$$\text{OP\_MUL}(\varphi_M, \varphi_A, \varphi_I) = (A_0 * A_1 \cdot A_2 \cdot \ldots \cdot A_k, \varphi_A, \varphi_I),$$

where $*$ signifies the multiplication of integer numbers. Now we can prove that using OP_MUL we can create scripts that can not be evaluated in polynomial time. The following lemma gives an insight as to how this is possible.

Lemma 4.1. There exists a script $S = f_0 \cdot \ldots \cdot f_n \in (\{\text{OP\_PUSH}_2, \text{OP\_MUL}, \text{OP\_DUP}\})^*$, *such that*

$$(f_n \circ \ldots \circ f_0)(\varepsilon, \varepsilon, \varepsilon) = (A, \varepsilon, \varepsilon)$$

and $A \geq 2^{2^{cn}}$, for some c > 0.

Proof. For this, consider the script

$$S = \text{OP\_PUSH}_2 \cdot \text{OP\_DUP} \cdot \text{OP\_MUL} \cdot \text{OP\_DUP} \cdot \text{OP\_MUL} \cdot \ldots \cdot \text{OP\_DUP} \cdot \text{OP\_MUL}.$$

The main idea of the proof is that this script will end its execution with 2^{2^m} at the top of the main stack, where m is the number of repetitions of the OP_DUP·OP_MUL sequence of operators.

More formally, let $m \in \mathbb{N}$ be an arbitrary number and $S_m = \text{OP\_PUSH}_2 \cdot (\text{OP\_DUP} \cdot \text{OP\_MUL})^m$. Then:

$$S_m(\varepsilon, \varepsilon, \varepsilon) = (2^{2^m}, \varepsilon, \varepsilon).$$

This can be easily shown by mathematical induction on m.

Base case. Consider $S_0 = \text{OP\_PUSH}_2$. Then we trivially have

$$S_0(\varepsilon, \varepsilon, \varepsilon) = (2, \varepsilon, \varepsilon) = (2^1, \varepsilon, \varepsilon) = (2^{2^0}, \varepsilon, \varepsilon).$$

Inductive step. Suppose that for an arbitrary number $i \in \mathbb{N}$, we have that $S_i(\varepsilon, \varepsilon, \varepsilon) = (2^{2^i}, \varepsilon, \varepsilon)$. Now, we can note that $S_{i+1} = S_i \cdot (\text{OP\_DUP} \cdot \text{OP\_MUL})$. Thus, we have that

$$\begin{aligned} S_{i+1}(\varepsilon, \varepsilon, \varepsilon) &= (\text{OP\_MUL} \circ \text{OP\_DUP})(S_i(\varepsilon, \varepsilon, \varepsilon)) \\ &= (\text{OP\_MUL} \circ \text{OP\_DUP})(2^{2^i}, \varepsilon, \varepsilon) \\ &= (2^{2^i} * 2^{2^i}, \varepsilon, \varepsilon) \\ &= (2^{2^{i+1}}, \varepsilon, \varepsilon). \end{aligned}$$

From Lemma 4.1 we can conclude that it is possible to construct a stack that has an element that is double exponential in magnitude and therefore exponential in size compared to the amount of operators in the script that is being executed[5]. Moreover, as each used operator is constant in size, the size of the constructed element is also exponential in the size of the script. This means that it is not possible to write, save or use such an element in polynomial time compared to the size of our script. Thus, such a script could not be evaluated in polynomial time. A similar result can be obtained for some other operators that were supported in the original proposal of Script, such as for instance OP_CAT. In light of this result, disabling some operators seems well justified [see (Bitcoin Wiki, 2021) for the full list of disabled operators].

Taking this result into consideration, it is important to prove that with the current definition of Script, any script can be evaluated in polynomial time. Letting Ptime be the class of problems that can be solved in polynomial time, we have the following:

Theorem 4.2. The problem Evaluation of Script is in Ptime.

Proof. The proof proceeds in four steps. First, we show that Script operators, by themselves, do not introduce transformations that produce drastic changes in the stack. We then show that this remains true when analyzing sequences of operators. This, in turn, allows us to show that the execution time of a script over an empty stack is actually in polynomial time, from which the proof of this theorem readily follows.

We start by defining three auxiliary functions that will help us establish some properties over the execution of operators: function maxelem: $\mathbb{Z}^* \times \mathbb{Z}^* \rightarrow \mathbb{N}$ obtains the size of the biggest element in a stack (independent of its sign), function elemnr: $\mathbb{Z}^* \rightarrow \mathbb{N}$ obtains the amount of elements in a stack, and maxpush: $O^* \rightarrow \mathbb{N}$ provides the maximum element pushed by a script, independent of its sign.

Let $S = f_0 \cdot \ldots \cdot f_n \in O*$, $\varphi_M = A_0 \cdot \ldots \cdot A_k \in \mathbb{Z}^*$ and $\varphi_A = B_0 \cdot \ldots \cdot B_\ell \in \mathbb{Z}^*$. We define the values of these functions as follows;

$$\begin{aligned}
\text{maxelem}(\varphi_M, \varphi_A) &= \max_{A \in \{A_0,\ldots,A_k\} \cup \{B_0,\ldots,B_\ell\}} |A| \\
\text{elemnr}(\varphi) &= k+1 \\
\text{maxpush}(S) &= \max\{|C| : \text{OP\_PUSH}_C \in \{f_0,\ldots,f_n\}\}
\end{aligned}$$

As the reader might have already noticed, empty stacks provides for edge cases in which maxelem is not defined, and likewise for maxpush, and so they must be accounted for separately. To that extent, let $S' = g_0 \cdot \ldots \cdot g_m \in O*$ such that $\{g_0,\ldots,g_m\} \cap \{\text{OP\_PUSH}_C \mid C \in \mathbb{Z}\} = \varnothing$. Then we have that:

$$\begin{aligned}
\text{maxelem}(\varepsilon, \varepsilon) &= 0 \\
\text{maxpush}(S') &= 0
\end{aligned}$$

As we mentioned in our proof strategy, our first task is to show that none of the operators in Script produces a stack that explodes in size. In formal terms, this translates to restrictions for the auxiliary functions introduced previously.

Lemma 4.3. Let $f \in O$, $\varphi_M \in \mathbb{Z}^*$, $\varphi_A \in \mathbb{Z}^*$ and $\varphi_I \in \{0,1\}*$ such that $f(\varphi_M, \varphi_A, \varphi_I) = (\varphi'_M, \varphi'_A, \varphi'_I)$. Then we have that:

$$\text{elemnr}(\varphi'_M) \leq \text{elemnr}(\varphi_M) + 3$$

Moreover, assume that $f \notin \{\text{OP\_DEPTH}, \text{OP\_HASH160}\} \cup \{\text{OP\_PUSH}_C \mid C \in \mathbb{Z}\}$. Then we have

$$\text{maxelem}(\varphi'_M, \varphi'_A) \leq 2 \cdot \text{maxelem}(\varphi_M, \varphi_A) + 1$$

Proof. For this proof one needs a one-by-one analysis showing that each of the operators of Script satisfies these bounds. We give a few examples of how this is proved for some particular operators, the full details for the entire Script language are provided as **Supplemental Material**.

We show how to prove the bound on elemnr using the OP_3DUP operator. For an arbitrary pair of stacks $\varphi_M = A_0 \cdot \ldots \cdot A_k \in \mathbb{Z}^*$ and $\varphi_A \in \mathbb{Z}^*, \varphi_I \in \{1\}^*$, if these stacks fulfill the conditions for the OP_3DUP operator, then we have that

$$\begin{aligned}
\text{OP\_3DUP}(\varphi_M, \varphi_A, \varphi_I) &= (\varphi'_M, \varphi'_A, \varphi'_I) \\
&= (A_0 \cdot A_1 \cdot A_2 \cdot \varphi_M, \varphi_A, \varphi_I),
\end{aligned}$$

or, in other words, OP_3DUP pushes elements A_0, A_1 and A_2 onto φ_M. This means that $\text{elemnr}(\varphi'_M) = \text{elemnr}(\varphi_M) + 3$, which was to be shown.

For the bound on the size maxelem of elements in the stack, we use the OP_ADD operator to illustrate the proof. Again, for an arbitrary pair of stacks $\varphi_M = A_0 \cdot \ldots \cdot A_k \in \mathbb{Z}^*, \varphi_A \in \mathbb{Z}^*, \varphi_I \in \{1\}^*$, if these stacks fulfill the conditions for the OP_ADD operator, then we have that

$$\begin{aligned}
\text{OP\_ADD}(\varphi_M, \varphi_A, \varphi_I) &= (\varphi'_M, \varphi'_A, \varphi'_I) \\
&= ((A_0 + A_1) \cdot A_2 \cdot \ldots \cdot A_k, \varphi_A, \varphi_I).
\end{aligned}$$

It is clear that $|A_0|, \ldots, |A_k| \leq \text{maxelem}(\varphi_M)$, hence we have that

$$|A_0| + |A_1| \leq 2 \cdot \text{maxelem}(\varphi_M) \leq 2 \cdot \text{maxelem}(\varphi_M) + 1.$$

By combining both results we can conclude that

$$\text{maxelem}(\varphi'_M) \leq 2 \cdot \text{maxelem}(\varphi_M) + 1,$$

which was to be shown. ■ □

Our next step is to use the bounds presented above to provide upper bounds on the values of these functions over the execution of complete scripts. These bounds must take into account the size of the stacks, so we need to discuss how these are encoded. For our

[5]In reality, Script has constraints over the size that its elements can occupy. Thus, if a number were to surpass this limit, an exception would be raised. However, we feel that it is important to work with a generalization of the language that does not constrain the size of the elements because the maximum size that is imposed is much larger than what a user would normally interact with.

complexity results, we assume that stacks are represented as arrays of integer elements. Assuming that the elements in the stacks are represented in binary notation, and that we use one extra bit to represent the sign of each number, we define the size $\|\varphi\|$ of a stack $\varphi = A_0 \cdot \ldots \cdot A_k \in \mathbb{Z}^*$ as:

$$\|\varphi\| = \sum_{i=0}^{k} \log_2 (|A_i| + 1).$$

From this definition, we can derive the following bounds:

$$\log_2 (\text{maxelem}(\varphi, \varepsilon) + 1) \leq \|\varphi\| \tag{3}$$

$$\text{elemnr}(\varphi) \leq \|\varphi\| \tag{4}$$

These bounds will help us in relating several results that make use of the auxiliary functions with the size of the representation of the stack. This is useful because we will be interested in establishing a relationship between the runtime of executing a script with an initial stack and the sizes of the inputs.

Let us now turn to (upper) bound the values of the auxiliary functions. In particular, we need to prove that the different elements of the stacks, during the execution of a script, are of polynomial-size in the sizes of the script and the initial stack. This idea is formalized in the following lemma; the proof is once again by a direct examination of each operator.

Lemma 4.4. Let $S = f_0 \cdot \ldots \cdot f_n \in O*$ and $\varphi_M \in \mathbb{Z}^*$. After any partial execution $(f_i \circ \ldots \circ f_0)(\varphi_M, \varepsilon, \varepsilon)$, the following conditions hold.

1. The amount of elements that can appear in the main stack is bounded by $\text{elemnr}(\varphi_M) + 3(n + 1)$.
2. The biggest element that can appear in either stack is bounded by

$$p_{\text{maxelem}} (2^n, 2^{\|\varphi_M\|}, \text{maxpush}(S)),$$

for some fixed polynomial p_{maxelem}.

3. The size of the representation of the main stack is bounded by

$$p_{size} (n, \|\varphi_M\|, \log_2 (\text{maxpush}(S) + 1)),$$

for some fixed polynomial p_{size}.

We finally have all the ingredients to provide the upper bound on the execution time of script evaluation. For this result we consider a naïve algorithm that receives as input a script $S = f_0 \cdot \ldots \cdot f_n \in O*$ and a stack $\varphi \in \mathbb{Z}^*$ and executes each operator in sequence over the stack. This is without loss of generality, as using faster, more involved algorithms can only decrease the total running time of the operations. In what follows we use T to denote the execution time of said algorithm, having in mind that the full execution involves working over a trio of stacks (the main stack, the alt-stack and the auxiliary stack for the IF-ELSE control flow). Therefore, we treat T as a function $T\colon O^* \times \mathbb{Z}^* \times \mathbb{Z}^* \times \{0, 1\}^* \to \mathbb{N}$.

Lemma 4.5. Let $S = f_0 \cdot \ldots \cdot f_n \in O*$ and $\varphi \in \mathbb{Z}^*$. Then we have that

$$T (S, \varphi, \varepsilon, \varepsilon) \leq p_T (n, \|\varphi\|, \log_2 (\text{maxpush}(S) + 1))$$

for some fixed polynomial p_T (independent of S and φ).

The proof of this lemma is by induction, using Lemma 4.4 and Lemma 4.3 for the inductive and base cases.

Having established that the execution time of the algorithm that applies a script to an initial stack can be executed in polynomial time, we can move on to prove that script evaluation is in PTIME. As we have previously discussed, an algorithm that performs script evaluation starts by executing an unlocking script over a trio of empty stacks and then, if the final control stack is empty, it executes a locking script over the previous final main stack and two empty stacks. Thus, all we need to do is to show that both of these operations take polynomial time when executed sequentially. This is shown in the following lemma. Note that we only have consider the case in which the unlocking script that the algorithm receives does not result in an error and that finishes with an empty control stack; the remaining cases imply an early stop in the algorithm so they are also captured by the bounds for a complete execution.

Lemma 4.6. Let $S_L = f_0 \cdot \ldots \cdot f_n \in O*$ and $S_U = g_0 \cdot \ldots \cdot g_m \in O*$, such that

$$(g_m \circ \ldots \circ g_0)(\varepsilon, \varepsilon, \varepsilon) = (\varphi_M, \varphi_A, \varepsilon).$$

Then, there is a fixed polynomial p_{emp} (independent of S_L and S_U) such that:

$$T (S_U, \varepsilon, \varepsilon, \varepsilon) + T (S_L, \varphi_M, \varepsilon, \varepsilon) \leq p_{emp} (m, n, \log_2 (\text{maxpush}(S_U) + 1), \log_2 (\text{maxpush}(S_L) + 1))$$

Proof. Let $S_L = f_0 \cdot \ldots \cdot f_n \in O*$ and $S_U = g_0 \cdot \ldots \cdot g_m \in O*$, such that

$$(g_m \circ \ldots \circ g_0)(\varepsilon, \varepsilon, \varepsilon) = (\varphi_M, \varphi_A, \varepsilon).$$

From Lemma 4.5 there is a fixed polynomial p_T and the following bound on $T(S_U, \varepsilon, \varepsilon, \varepsilon)$:

$$\begin{aligned} T (S_U, \varepsilon, \varepsilon, \varepsilon) &\leq p_T (m, \|\varepsilon\|, \log_2 (\text{maxpush}(S_U) + 1)) \\ &\leq p_U (m, \log_2 (\text{maxpush}(S_U) + 1)), \end{aligned}$$

where p_U is again a fixed polynomial. We can also bound $T(S_L, \varphi_M, \varepsilon, \varepsilon)$ in the same way:

$$T (S_L, \varphi_M, \varepsilon, \varepsilon) \leq p_T (n, \|\varphi_M\|, \log_2 (\text{maxpush}(S_L) + 1)).$$

Moreover, from Lemma 4.4 we can also bound $\|\varphi_M\|$:

$$\|\varphi_M\| \leq p_{size} (m, \|\varepsilon\|, \log_2 (\text{maxpush}(S_U) + 1)).$$

By combining both of these bounds we can conclude that

$$T (S_L, \varphi_M, \varepsilon, \varepsilon) \leq p_L (m, n, \log_2 (\text{maxpush}(S_U) + 1), \log_2 (\text{maxpush}(S_L) + 1)),$$

for some polynomial p_L. Furthermore, adding equations for both executions over S_L and S_U, we obtain:

$$\begin{aligned}T(S_U,\varepsilon,\varepsilon,\varepsilon) + T(S_L,\varphi_M,\varepsilon,\varepsilon) &\le p_U(m,\log_2(\text{maxpush}(S_U)+1))\\ &\quad + p_L(m,n,\log_2(\text{maxpush}(S_U)+1),\log_2(\text{maxpush}(S_L)\\ &\quad +1)) \le p_{emp}(m,n,\log_2(\text{maxpush}(S_U)+1),\log_2(\text{maxpush}(S_L)\\ &\quad +1)),\end{aligned}$$

where p_{emp} is a fixed polynomial independent of S_L and S_U. ■ □

Note that we have not included the validity checks that need to be performed between the executions of both scripts and also at the end of the execution of the locking script to determine whether the execution was successful. This is because these checks can be performed in constant time and do not impact the complexity analysis of the problem. This concludes the proof of the Theorem.

4.3 Unlockability is Computationally Infeasible

The evaluation of a pair of scripts can be done efficiently, but what about checking whether a script is unlockable? Recall that the Unlockability of Script problem receives a locking script l, and consists of checking whether there exists any unlocking script u such that l and u result in a positive answer when evaluated together.

Unfortunately, the following result tells us that this problem cannot be solved efficiently.

Theorem 4.7. The problem Unlockability of Script is NP-hard.

Proof. To prove the theorem, we provide a polynomial-time reduction from 3SAT, which is a well-known NP-complete problem. Let $\psi = C_0 \wedge C_1 \wedge \ldots \wedge C_m$ be a formula in CNF, where each clause C_i is the conjunction of three literals:

$$C_i = u_{i,0} \vee u_{i,1} \vee u_{i,2},$$

that is, each $u_{i,j}$ is either a propositional variable x or the negation of a propositional variable ¬x. Besides, assume that $\{x_0, x_1, \ldots, x_k\}$ is the set of variables occurring in ψ. Next, we show how to construct in polynomial-time a script l_ψ that can be used to check whether ψ is satisfiable.

By definition of the problem Unlockability of Script, the script l_ψ has to be executed over the main stack resulting from the execution of an unlocking script. Thus, l_ψ interprets such a stack as a truth assignment for the formula ψ, and verifies whether such an assignment satisfies this formula. In this way, an unlocking script for l_ψ represents a truth assignment satisfying ψ, so that ψ is satisfiable if and only if l_ψ is unlockable. More precisely, we define l_ψ as follows:

$$l_\psi := l_{\text{length}} \cdot l_{\text{binary}} \cdot l_{\text{sat}} \cdot l_{\text{end}}$$

where each script l_{length}, l_{binary}, l_{sat}, l_{end} are as defined below.

- The script l_{length} checks whether the stack that it receives as input contains k + 1 elements (which is the number of variables occurring in ψ):

$$l_{\text{length}} := \text{OP\_DEPTH} \cdot \text{OP\_PUSH}_{k+1} \cdot \text{OP\_EQUALVERIFY}$$

- The script l_{binary} verifies whether each one of the k + 1 elements of the stack is either 0 or 1. More precisely, we have that:

$$l_{\text{binary}} := l^0_{\text{binary}} \cdot l^1_{\text{binary}} \cdot \ldots \cdot l^k_{\text{binary}},$$

where for every i ∈ {0, 1, . . ., k}:

$$l^i_{\text{binary}} := \text{OP\_PUSH}_i \cdot \text{OP\_PICK} \cdot \text{OP\_IFDUP} \cdot \text{OP\_IF} \cdot \text{OP\_PUSH}_1 \cdot \text{OP\_EQUALVERIFY} \cdot \text{OP\_ENDIF}.$$

- The script l_{sat} checks whether the formula ψ is satisfied by the truth assignment stored in the stack, that is, by the sequence of k + 1 symbols 0 and 1 stored in the stack. More precisely, we have that:

$$l_{\text{sat}} := l^0_{\text{sat}} \cdot l^1_{\text{sat}} \cdot \ldots \cdot l^m_{\text{sat}},$$

where each script l^i_{sat} ($0 \le i \le m$) is defined as follows. Assuming that $C_i = u_{i,0} \vee u_{i,1} \vee u_{i,2}$, we have that:

$$l^i_{\text{sat}} := l^{i,0}_{\text{val}} \cdot l^{i,1}_{\text{val}} \cdot l^{i,2}_{\text{val}} \cdot l_{\text{add}},$$

where $l^{i,0}_{\text{val}}$, $l^{i,1}_{\text{val}}$, $l^{i,2}_{\text{val}}$ and l_{add} are defined as follows. If $u_{i,0} = x_r$ for some r ∈ {0, . . ., k}, then

$$l^{i,0}_{\text{val}} := \text{OP\_PUSH}_r \cdot \text{OP\_PICK}.$$

Thus, $l^{i,0}_{\text{val}}$ puts the value assigned to the propositional variable x_r in the top of the stack. On the other hand, if $u_{i,0} = \neg x_r$ for some r ∈ {0, . . ., k}, then

$$l^{i,0}_{\text{val}} := \text{OP\_PUSH}_1 \cdot \text{OP\_PUSH}_{r+1} \cdot \text{OP\_PICK} \cdot \text{OP\_SUB}.$$

Hence, $l^{i,0}_{\text{val}}$ puts the value assigned to $\neg x_r$ in the top of the stack (this value is obtained by subtracting the value assigned of x_j from 1). Similarly, if $u_{i,1} = x_s$ for some s ∈ {0, . . ., k}, then

$$l^{i,1}_{\text{val}} := \text{OP\_PUSH}_{s+1} \cdot \text{OP\_PICK}.$$

Thus, $l^{i,1}_{\text{val}}$ puts the value assigned to x_s in the top of the stack. Notice that the operator OP_PUSH_{s+1} has to be used as the stack contains the extra value computed by the script $l^{i,0}_{\text{val}}$. In the same way, if $u_{i,1} = \neg x_s$ for some s ∈ {0, . . ., k}, then

$$l^{i,1}_{\text{val}} := \text{OP\_PUSH}_1 \cdot \text{OP\_PUSH}_{s+2} \cdot \text{OP\_PICK} \cdot \text{OP\_SUB},$$

so that the value assigned to $\neg x_s$ is put in the top of the stack by $l^{i,1}_{\text{val}}$. Moreover, $l^{i,2}_{\text{val}}$ is defined in the same way but considering that the stack contains the extra values computed by the scripts $l^{i,0}_{\text{val}}$ and $l^{i,1}_{\text{val}}$. Finally,

$$l_{\text{add}} := \text{OP\_ADD} \cdot \text{OP\_ADD} \cdot \text{OP\_VERIFY},$$

which allows us to add the values for the three literals in the clause. If at least one of them is positive, the result is greater than 0 and the execution of OP_VERIFY is successful. Otherwise, the result is 0 and the execution of OP_VERIFY fails.

- Finally, we have that:

$$l_{\text{end}} := \text{OP\_PUSH}_1,$$

which ensures that if l_{length}, l_{binary} and l_{sat} are executed successfully, then the top element in the main stack is 1.

From the definition of l_ψ, it is straightforward to prove that ψ is satisfiable if and only if l_ψ is unlockable, and that l_ψ can be constructed in polynomial time in the size of ψ. Hence, we have provided a polynomial-time reduction form 3SAT to the problem UNLOCKABILITY OF SCRIPT, thus showing that the latter is NP-hard.

5 DISCUSSION AND FUTURE WORK

In this work, we focused on Script, the scripting language of the Bitcoin protocol, and contribute to its understanding in three aspects:

- First, we provided a formal mathematical model for Script, which we used to study its algorithmic properties and main characteristics.
- Second, we (re)prove the folklore result stating that Script can be evaluated in PTIME.
- And third, we showed that determining whether a script is unlockable is NP-hard.

These three advancements allow us to better understand Script, and provide some insight into the behavior of nodes on the Bitcoin network. First, we observe that the vast amount of scripts used in Bitcoin transactions only establish the most basic unlocking conditions. Intuitively, one of the main reasons for this is that the nodes in the network tend to favor standard locking scripts, because they guarantee that their executions will be short and efficient. Our formalization, together with the result on efficiently evaluating Script, actually tell us that this might be somewhat overly cautious, given that any Bitcoin script can be run efficiently by a node. On the other hand, if we are preoccupied with detecting unspendable outputs, for instance to remove them from the unspent transaction output (UTXO) pool, then the NP-hardness result tells us that sticking to standard scripts is indeed a safe tactic, since no efficient algorithm exists for checking whether an output is spendable (unless P= NP).

Looking ahead, we believe that further investigation into unlockability is necessary. As mentioned previously, unlockability is useful for two reasons: 1) a wallet creating a transaction would definitely want to discard unspendable outputs, or at least warn the user about them; and 2) network nodes would want to remove unspendable outputs form their UTXO pool. The NP-hardness result tells us that this, in principle, will not be possible. However, if we were able to also show that unlockability can be solved in NP, a SAT solver might be used to check Script unlockability. At this point in time, SAT solvers have advanced to the point that it is feasible to determine whether a formula is satisfiable for reasonable inputs, so that we might be able to use these techniques to check structural consistency of a Script (provided that the unlockability problem belongs to NP). Of course, here we would need to assume, as in any Bitcoin transaction, that the correct cryptographic data is provided by the recipient, thus allowing us to verify whether the Script has any logical errors using the SAT solver. Our conjecture at this point is that the unlockability problem indeed belongs to NP.

Another direction worth pursuing would be to look for tractable fragments of Script in terms of the unlockability problem. Moreover, our formalization also allows to check whether a specific property is expressible using Script, which might be of interest when exploring smart contracts that could potentially be supported. Overall, we hope to make this work useful to the users wanting to specify non-trivial spending conditions, ultimately making the usage of non-standard scripts a more accepted practice.

DATA AVAILABILITY STATEMENT

The original contributions presented in the study are included in the article/**Supplementary Material**, further inquiries can be directed to the corresponding author.

AUTHOR CONTRIBUTIONS

All authors listed have made an equal contribution to the work and approved it for publication.

FUNDING

This work was funded by ANID—Millennium Science Initiative Program—Code ICN17_002, and by Fondecyt grant 1191337.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fbloc.2021.770503/full#supplementary-material

REFERENCES

Andrychowicz, M., Dziembowski, S., Malinowski, D., and Mazurek, Ł. (2014). "Modeling Bitcoin Contracts by Timed Automata," in Formal Modeling and Analysis of Timed Systems - 12th International Conference, FORMATS 2014, Florence, Italy, September 8-10, 2014. Editors A. Legay and M. Bozga (Springer), 7–22. Proceedings of Lecture Notes in Computer Science. doi:10.1007/978-3-319-10512-3_2

Antonopoulos, A. M. (2017). *Mastering Bitcoin: Programming the Open Blockchain*. (Sebastopol, CA: O'Reilly Media, Inc.)

Antonopoulos, A. M., and Wood, G. (2018). *Mastering Ethereum: Building Smart Contracts and Dapps*. (Sebastopol, CA: O'Reilly Media, Inc.).

Atzei, N., Bartoletti, M., and Cimoli, T. (2017). "A Survey of Attacks on Ethereum Smart Contracts (Sok)," in Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017 (Springer), 164–186.

Proceedings of Lecture Notes in Computer Science. doi:10.1007/978-3-662-54455-6_8

Atzei, N., Bartoletti, M., Cimoli, T., Lande, S., and Zunino, R. (2018a). "Sok: Unraveling Bitcoin Smart Contracts," in Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, GreeceApril 14-20, 2018 (Springer), 217–242. Proceedings of Lecture Notes in Computer Science. doi:10.1007/978-3-319-89722-6_9

Atzei, N., Bartoletti, M., Lande, S., and Zunino, R. (2018b). "A Formal Model of Bitcoin Transactions," in Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26-March 2, 2018 (Springer), 541–560. Revised Selected Papers. doi:10.1007/978-3-662-58387-6_29

Bartoletti, M., and Zunino, R. (2018). "Bitml: A Calculus for Bitcoin Smart Contracts," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018 (ACM), 83–100.

Bartoletti, M., and Zunino, R. (2019). Formal Models of Bitcoin Contracts: A Survey. *Front. Blockchain* 2, 8. doi:10.3389/fbloc.2019.00008

Bitcoin Wiki (2021). Bitcoin Wiki - Script. [Dataset]. Available at: https://en.bitcoin.it/wiki/Script (Accessed February 15, 2021).

Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J. A., and Felten, E. W. (2015). "Sok: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies," in 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015, 104–121. doi:10.1109/sp.2015.14

Buterin, V. (2014). A next-generation smart contract and decentralized application platform*White Paper* 3 (37).

Dannen, C. (2017). *Introducing Ethereum and Solidity*, Vol. 318. (Berkeley, CA: Springer).

Github (2010). Script Implementation: Security Improvements. [Dataset]. Available at: https://github.com/bitcoin/bitcoin/commit/6ff5f718b6a67797b2b3bab8905d607ad216ee21 (Accessed February 15, 2021).

Jansen, M., Hdhili, F., Gouiaa, R., and Qasem, Z. (2019). "Do smart Contract Languages Need to Be Turing Complete," in Blockchain and Applications - International Congress, BLOCKCHAIN 2019, Avila, Spain, 26-28 June, 2019 (Springer), 19–26. doi:10.1007/978-3-030-23813-1_3

Klomp, R., and Bracciali, A. (2018). "On Symbolic Verification of Bitcoin's Script Language," in Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2018 International Workshops, DPM 2018 and CBT 2018, Barcelona, Spain, September 6-7, 2018 (Springer), 38–56. Proceeding of Lecture Notes in Computer Science. doi:10.1007/978-3-030-00305-0_3

Nakamoto, S. (2008). Bitcoin: A Peer-To-Peer Electronic Cash System. Technical Report, Manubot.

Narayanan, A., Bonneau, J., Felten, E. W., Miller, A., and Goldfeder, S. (2016). *Bitcoin and Cryptocurrency Technologies - A Comprehensive Introduction*. (Princeton, NJ: Princeton University Press).

O'Connor, R. (2017). "Simplicity: A New Language for Blockchains," in Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2017, Dallas, TX, USA, October 30, 2017 (ACM), 107–120.

Singh, A., Parizi, R. M., Zhang, Q., Choo, K.-K. R., and Dehghantanha, A. (2020). Blockchain Smart Contracts Formalization: Approaches and Challenges to Address Vulnerabilities. *Comput. Security* 88, 101654. doi:10.1016/j.cose.2019.101654

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.