On symbolic verification of Bitcoin's SCRIPT language

Rick Klomp and Andrea Bracciali [*]

Computing Science and Mathematics, University of Stirling, UK
{name.surname}@stir.ac.uk

Abstract. Validation of Bitcoin transactions rely upon the successful execution of scripts written in a simple and effective, non-Turing-complete by design language, simply called SCRIPT. This makes the validation of *closed* scripts, i.e. those associated to actual transactions and bearing full information, straightforward. Here we address the problem of validating *open* scripts, i.e. we address the validation of redeeming scripts against the whole set of possible inputs, i.e. under which general conditions can Bitcoins be redeemed? Even if likely not one of the most complex languages and demanding verification problems, we advocate the merit of formal verification for the Bitcoin validation framework. We propose a symbolic verification theory for of *open* SCRIPT, a verifier tool-kit, and illustrate examples of use on Bitcoin transactions. Contributions include 1) a formalisation of (a fragment of) the language; 2) a novel symbolic approach to SCRIPT verification, suitable, e.g. for the verification of newly defined and non-standard payment schemas; and 3) building blocks for a larger verification theory for the developing area of Bitcoin smart contracts. The verification of smart contracts, i.e. agreements built as transaction-based protocols, is currently a difficult to formalise and computationally demanding problem.

1 Introduction

The Bitcoin framework [15] enable monetary transactions of a virtual currency amongst untrusted individuals. The construction relies on a novel interpretation of distributed consensus for the validation of a decentralised ledger recording the Bitcoin transactions. Interestingly, the validation of transactions in terms of their correctness, e.g. no double spending and proper ownership of the virtual coins, is demanded to the successful execution of cryptography-fenced scripts associated to transactions. Replicated execution of scripts is supported by a network of peers, whose consensus guarantees the validity of transactions.

Striving for correctness, robustness and efficiency in such an unconventional and constrained execution model, the scripting language SCRIPThas been designed according to minimality principles, e.g. it is not Turing complete, has no recursion, cycles or procedure calls, has an execution cost proportional to the length of the code, and "dangerous" operations like MUL are not allowed. However, even if most transactions exploit standard payment schemes based on simple scripts believed to be robust, free and

more complex payment schemas are allowed, new standard schema can be introduced, and, interestingly, a whole area of protocols and smart contracts based on transactions are being developed [6, 14].

Despite the apparent simplicity of SCRIPT in itself, such a scenario calls for formal verification of critical scripts validating financial transactions, which in the case of Bitcoin alone have a market cap of about 100B USD (2018 - about half the GDP of a country like Greece).

The general validation schema is composed by two scripts, an *input script* in charge of providing data and credential to authorise the transaction, and an *output script*, whose structure basically defines the validation scheme and is in charge of checking that the data provided do actually enable the transaction.

We address the problem of the *satisfiability of "open" output scripts*, i.e. under which general conditions an input script exists capable of providing the right information to let the output script successfully terminate and the transition be validated?

Although the simulation and execution of closed scripts present no problems and many tools and simulators are available, we observe that verification frameworks for satisfiability of open scripts are not so widespread and we believe that there are opportunities for further research in the area.

We introduce a symbolic verification framework which simulate the execution of an output script accumulating minimal constraints, akin to a lazy evaluation approach, for its successful termination. For each successful symbolic simulation, these constraints, if satisfiable, specify one (or more) possible input script that can provide enough information to redeem the transaction.

The contributions of this paper are:

(1) a (yet to be completed, but including major constructs) formal description of the SCRIPT language, which has been mostly presented informally and by code releases. We have focussed on salient features for the correctness of validation, often digging in the code for clarity;

(2) the novel, to the best of our knowledge, symbolic framework allowing us to derive a correct and (ongoing research) complete specification of all the possible, and possibly unintended, ways to redeem transactions. Beyond the simplicity by construction of SCRIPT, complex "non-standard" payment schema can and have been provided, worth being clearly analysed. Besides, new payment schemas can be defined and accepted as standard by the community, for instance to overcome existing limitations, similarly to P2PkH improving over the P2Pk schema. In such cases, a clear understanding of the implications of the schema in use and the novel ones is desirable;

(3) valid transactions are being used to define more articulated protocols in the context of *smart contracts* over the Bitcoin blockchain, e.g. self-enforcing agreements in the form of executable code [16]. We envisage the framework here presented as a building block of a larger verification framework aimed at the extra level of complexity introduced by smart contracts, whose verification is currently a difficult to formalise and computationally demanding open research problem. It is worth reminding how this area is prone to "simple", and supposedly well-understood, security failures, which easily lead to consistent financial loss, e.g. the recent overflow-based case of the Parity wallet [13].

In this paper we introduce the symbolic verification framework, SCRIPT ANAL-YSER, an open source application supporting the presented symbolic verification, and discuss two examples of non-standard transactions appeared in the Bitcoin blockchain, which are slightly more complex than a typical standard payment schema, and, we believe, illustrate how the proposed symbolic verification can support a better understanding of the solidity of Bitcoin's validation machinery.

For the sake of space, while we strived for a comprehensive presentation, a fully formal presentation is demanded to a forthcoming extended paper.

2   Bitcoin's blockchain and transactions

A quick review of the most relevant aspects of the design of Bitcoin are recapped here. The interested reader is referred to the SOK paper [10] for details.

The Bitcoin blockchain consists of a data structure implementing a distributed ledger. This can be understood, informally speaking, as an append only, and therefore immutable, list of blocks of data maintained by a peer-2-peer network. The blockchain is freely accessible and anyone can be a node (Bitcoin is a permission-less blockchain). Each node stores an identical copy of the ledger. Main innovation is that the ledger is *decentralised*, i.e. the responsibility for certifying the correctness of the ledger is shared amongst all the nodes, no one being in charge, and guaranteed by a cryptographically-supported distributed consensus, currently the *proof-of-work*. The whole network guarantees correctness, and at least half of the computational power in the network is needed in order to alter the ledger.

The ledger records payments amongst accounts, i.e. addresses, based on PKI: the address can be derived from the public key and the private key is used to prove ownership of the address and the crypto-money therein.

Transactions move Bitcoins from one address to another. Each transaction has potentially multiple *inputs*, i.e. it draws Bitcoins from multiple addresses, and delivers the drawn Bitcoins over potentially multiple *outputs*, i.e. pays multiple recipient addresses. Once moved into an address, Bitcoins are redeemable by a suitable subsequent transaction. Each input of a subsequent redeeming transaction needs to provide suitable credential in order to "spend" Bitcoins. This is done by an *input script*, or *SigScript*, whose execution provides credentials, which are then validated by the *output script* associated to the output of the previous transaction. The format of the output script is, in principle, free, although a few standard output scripts are commonly used. The successful execution of the input and then output script makes the transaction valid and it can be recoded on the ledger.

Bitcoin architecture uses *hash functions*, i.e. a mapping from an unbounded domain to a fixed domain, which is straightforward to compute but practically impossible to invert. An hash if often used to prove properties or validate a piece of information. Also *digital signature* are used, based on public key cryptography. A signature, used to *validly* sign a message, proves that the signer authorized this message. Alteration of the message invalidates the signature. Generating a valid signature is straightforward, when owing the private key, generally infeasible otherwise. Verifying the validity of the combination of signature, public key and message is straightforward.

3 Related work

Delmonino et-al [12], and Bartoletti and Pompianu [7], empirically analyse common patterns in designs of smart contracts. The authors show that some of the design patterns, though they are commonly applied, are actually undesired practices due to high odds of bug introducement. Increased risk of faults in (smart) contracts decreases their trustworthiness, as any fault possibly enables unexpected side effects (e.g. enabling a malicious party to claim, i.e. steal, honestly invested currency). As such, these results highlight the importance of improving smart contract design and verification practices.

Delgado-Segura et-al [11] present a tool (STATUS) for analysing Bitcoin's set of unspent transaction outputs (or UTXO). They present results from running this tool on the UTXO at block 491,868 of Bitcoin's blockchain (appended at October 26th, 2017). The UTXO was introduced to improve efficiency of validating new transactions. STATUS's main purpose is to analyse efficiency of the UTXO implementation approach, whereas our work aims to enable verification of certain properties of output SCRIPTs.

Andrychowicz et-al [6] introduce some interesting smart contracts and show through application of formal models that they are applicable in Bitcoin's ledger. In [5] they extend on this with a framework that captures the possible interaction sequences that may occur following a smart contract. Specifically, their approach enables automatic security verification by manually modeling the smart contract using timed automata. We propose a method which possibly enables automatically deriving a model, e.g. expressed in timed automatas, from only the output SCRIPTs of transactions. A generated model may then be further analysed, e.g. automatically following the approach in [6].

Lande and Zunino [14] propose a formal model to express smart contracts. They then use this formal model to survey and compare the smart contracts currently employed via Bitcoin's ledger. Furthermore, they propose designing bitcoin SCRIPTs using a high level language DSL that can guarantee security. We on the other hand attempt to derive security properties directly from the SCRIPT. Ultimately our goals, to increase quality of smart contracts, are similar however.

Bartoletti and Zunino [8] present BitML, a first high level DSL for designing smart contracts that are applicable through Bitcoin's ledger. The symbolic expressions that form an instance of BitML are easier to analyse than a Bitcoin's SCRIPT instance. Furthermore, they show that these BitML instances can be compiled to Bitcoin SCRIPTs.

Bhargavan et-al [9] propose verifying smart contracts written in Solidity (Ethereum's [17] primary smart contract language), by first either compiling Solidity code to F*, or decompiling EVM (Ethereum's virtual machine) bytecode to F*. It is then possible to verify whether the F* smart contract variant meets certain criteria by embedding these criteria into F*'s type and effect system.

4 SCRIPT - a fragment of

The validation of Bitcoin transactions reverts to the successful execution of two programs: an *input script* associated to the redeeming transaction, and an *output script* associated to a corresponding *unspent output* offered by an earlier transaction. Although the two scripts can be, and in some cases have been, freely defined, the intended behaviour is that the former provides credentials proving the ownership of the unspent

$$S ::= cmd; S \mid \diamond$$

$$cmd ::= \text{OP\_FALSE} \mid \text{OP\_TRUE} \mid \text{OP\_n} \qquad\qquad constants$$

$$\text{IF } S \mid \text{IFE } S \circ S \mid \text{VERIFY} \mid \text{OP\_RETURN} \mid \qquad flow\ control$$

$$\text{DUP} \mid \text{PUSH } d \mid \text{POP} \mid \qquad\qquad\qquad stack\ ops$$

$$\text{ADD} \mid \text{SUB} \mid \text{SIZE} \mid \qquad\qquad\qquad aritmetic\ ops$$

$$\text{EQ} \mid \text{LT} \mid \text{LTE} \mid \text{GT} \mid \text{GTE} \mid \text{AND} \mid \text{OR} \mid \qquad boolean\ ops$$

$$\text{CHKSIG} \mid \text{CHKMSIG} \mid \text{HASH256} \mid \text{HASH160} \mid \quad crypto\ ops$$

$$d ::= bs \mid Bool \mid int \mid key$$

Fig. 1. The syntax of SCRIPT.

coins, and the latter verifies them. The successful execution of the sequential composition of the two scripts validates the redeeming transaction, that can hence be accepted as valid and included in a block.

The script programming language consists of a stack-based programming language, called SCRIPT. It is not Turing-complete by design. It does not allow for cycles and has a restricted set of instructions. Furthermore, complexity of SCRIPT is also limited by transaction fees that are typically proportional to the space occupied by a transaction in a block, and hence to the length of its scripts. *Successful execution* reverts to termination, leaving the stack in a *true* state.

We provide here an informal description of the semantics of the language. Full details on the language can be found, e.g., in [1] by directly inspecting the source code of the Bitcoin Core client, or in [3] about the semantics of the SCRIPT instruction set, according to the core client's implementation.

The fragment of SCRIPT considered in this paper is described next, starting from the syntax in Figure 4. A program S in SCRIPT is either a sequence of stack operations (cmd) or the terminated program $\diamond$. Trailing and prefixed $\diamond$ are generally omitted or absorbed, respectively. Commands manipulate dynamically typed data on the stack, with automated type coercion. According to the execution model, operations are sequentially executed and may alter the stack by popping/pushing data. Operations may *fail* on missing data and mismatching or non-coercible types, causing a *runtime error*, immediately stopping the execution in a *failed* state. For instance, OP_RETURN fails by default, while ADD fails when the top of the stack does not contain two integers. The script *succeeds* if all the operations are successfully executed, i.e. they do not fail, *and* the (top of the) stack is *true*.

Commands are represented by mnemonic codes in Figure 4, while SCRIPT actually uses numeric opcodes. Our presentation of the semantics also introduces some simplifying abstractions.

Constants operations push data on the stack, e.g. OP_TRUE or OP_n which push the values $True$ and n, and never fail.

Flow control includes a branching command IF S, representing the conditional execution of the program S if the top stack value is true (it is popped when checked). SCRIPT would actually write it as the sequence IF; S; ENDIF. Analogously for the if-then-else construct IFE. Branching constructs may fail on empty stack when testing the condition, or, in our representation, because one of the commands in S fails. VERIFY pops the stack and then fails if the popped top of the stack was not true. OP_RETURN fails and, as a side effect, allows a limited amount of data to be recorded in the transaction, and hence permanently stored in the blockchain.

Stack operators work as expected. DUP and POP fail on empty stack. PUSH adds data on the stack.

SCRIPT operates on a stack with a 520-byte word. A *byte stream bs* is a sequence of bytes such that $|bs| \in [0, 520]$ ($|\_|$ is the length in bytes of a type), with ϵ the empty bs - which, e.g., can be successfully popped from the stack. bs_i ($bs_{[i,j]}$) represents a byte stream of length i (length in $[i, j]$), e.g. the result of a fixed-dimension hash operation.

A *boolean (Bool)* is a sequence of bytes with $|Bool| \in [0, 520]$ and 0, -0 and ϵ representing $False$.[1] An *integer (int)* is a 32-bit signed integer with $|int| \in [0, 4]$. $\top$ is the most generic type ($|\top| \in [0, 520]$). A *key (key)* is a sequence of bytes such that $|key| \in [9, 73]$. A public key pk can only be such that $|pk| = 65$ or $|pk| = 33$ (compressed). Some operations may have further requirements on keys, which will be modelled in the corresponding semantic rules. [2] [3] The type hierarchy is represented by (the transitive closure of) the following diagram, with solid lines for *being subtype of* and dotted ones representing allowed coercions under the constraint that byte length are respected, e.g. $b \in Bool$ can be successfully used as an integer only if $|b| \le 4$. $\top$, bs and $Bool$ are actually equivalent, while int and key disjoint. We introduced separate type entities for readability, and this hierarchy might be useful for more strictly typed variants of SCRIPT.

---
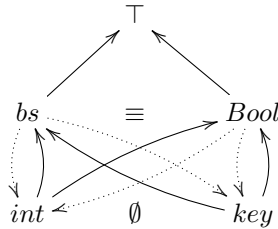
[1] Although there are multiple values equal to $false$ and multiple values equal to $true$, the Bitcoin Core client always instantiates these as [] and [0x01] respectively. Our symbolic verifier assumes the same representations.

[2] It is important to remark here that some type definitions, and other features, may depend on how the Bitcoin client is initialised. For instance, checking the dimension of a key depends on an initialisation parameter. We assume in this paper that the checking is done. We defer the verification against different possible initialisations to future work, noting that it must be addressed as different settings can affect correctness in different ways.

[3] Some operations may be more restrictive on the length of accepted keys, as well as their format. We will model this in the specific rules defining such operations, as appropriate. See Section 5.2.

A type B can be automatically coerced to a type A ($A \rhd B$):

$$\frac{|Bool| \leq 4}{int \rhd Bool} \text{ B2i} \qquad \frac{|\top| \leq 4}{int \rhd \top} \text{ T2i} \qquad \frac{|bs| \leq 4}{int \rhd bs} \text{ bs2i} \qquad \frac{|bs| \in \{33, 67\}}{key \rhd bs} \text{ bs2k}$$

Arithmetical operators ADD and SUB pop two int from the stack and push the result r. It is worth noting that over/underflow may cause $|r| = 5$, which may cause a subsequent type error, and the choice of some constants may be implementation-dependent, e.g. 0 can be ϵ, 0_4, 0_3, -0_4, $\dots$. Arithmetical operators fail on lack of data or type error, e.g. an incoercible bs_{10}.

Boolean operators work similarly, as expected, and fail on lack of data on the stack (any data can be interpreted as a boolean). VERIFY pops a boolean value from the stack, and verifies that it is equal to *true*. If this constraint is not met evaluation of the SCRIPT fails. Furthermore, the operation fails on lack of data. It is worth remarking here that some operators amongst those that push a boolean result on the stack have a _VERIFY variant, the so-called *verify operators*, such as EQ_VERIFY, CHECK_SIG_VERIFY. These variants are semantically equivalent to appending VERIFY after the original operator. For example, EQ_VERIFY is semantically equivalent to EQ;VERIFY.

Cryptographic operators check signatures, multi-signatures and push computed hash values on the stack. CHECK_SIG pops a *sig_key* s_k and a *pub_key* p_k from the stack, and checks if s_k is a valid signature of p_k combined with the hash of (part of) the transaction's data. It pushes the result of this check, i.e. *true* or *false*, to the stack. The non-VERIFY variant fails on lack of data and on type error. Note that performing CHECKSIG without a VERIFY does not enforce prior operations (including those defined by the input SCRIPT) to reach a matching signature check. This is only enforced when the result of CHECKSIG is constrained to *true* by a subsequent VERIFY operation. CHECK_MULTISIG checks the validity of a list of signatures against a list of public keys. First, an integer n_{sk} is popped, defining the number of provided signatures. Then n_{sk} signatures (sks) are popped. Similarly, an integer n_{pk} determining the number of provided public keys and n_{pk} public keys (pks) are popped. As a result, CHECK_MULTISIG pushes *true* on the stack if each signature is valid for a public key combined with the hash of (part of) the transaction, i.e. $\forall sk \in sks. \exists pk \in pks$ such that CHECK_SIG on (sk, pk) is *true*[4]. Otherwise, *false* is pushed on the stack. The non-VERIFY variant fails on lack of data and on type error. HASH operators pop a value from the stack and push the hash of this value on the stack. Hashes fail on lack of data.

Locktime operators add time constraints to redeeming Bitcoins. We mention them here for the sake of a more complete overview of SCRIPT but do not address them in this work.

Example 1. The SCRIPT program

<div align="center">DUP;POP;POP;DUP</div>

requires a top element in the stack, duplicates it, then pops the two copies, and then requires the presence of a second element initially on the stack and duplicates it.

---

[4] With the added note that matching public keys must be provided in the same order as the signatures they match with. Additionally, each provided public key can at most be matched with one signature.

Example 2. As an example of an unredeemable output SCRIPT, consider the program

$$\text{PUSH 0xFB15AC2030FB; ADD}$$

ADD will require two int values, one of which must be pushed to the stack by the input script. Regardless of what stack is left by the input SCRIPT, ADD will always fail on type error as the first value it pops (originating from the PUSH operation) is of type bs_6, which cannot be coerced to int.

5 Symbolic evaluation of SCRIPT programs

Given the SCRIPT code of an output script, its execution is simulated from an empty stack. Required data on the initial stack for a successful computation is defined via a lazy approach returning the weakest constraints on data for successful termination.

5.1 The execution stack model

A *symbolic stack* SK is an infinite (in both directions) list of indexed typed data, with indexes in $[-\infty, \infty]$ that uniquely identify a position and a datum in the stack:

$$[\dots d_{i+1}, d_i, d_{i-1} \dots]$$

$d_i = (x_i, t)$, with x_i a variable used to accumulate constraints on the expected data in the $i-th$ position, and t a type.

Two extra indexes associated to SK delimit the segment of significant data in SK: the *head index* h identifies the current top of the stack, and the *floor index* f denotes the first available position where data can be provided by the input script, that is the first position below the current bottom of the significant segment in SK. Initially, each d_i is undefined (and irrelevant) and $h = 0$ and $f = 0$, i.e. the top element is 0 and the first position where the input script could have added data is also 0. Such "inital" state is called the *empty stack* and denoted as SK_ϵ. Note that f can only decrease. Each command may further specify requirements on data by accumulating constraints on the associated variable. At the end of the computation, data of interest will be represented by the constraint store associated to the variables in $[f + 1, 0]$, i.e. the data required to be provided by the input script for successful termination of the output script, if any. Symbolic expressions exp in the stack consist of constants d, variables x_i, operations $op \in \{+, -, <, \leq, =, \geq, >, \wedge, \vee\}$:

$$exp ::= exp \; op \; exp \mid x_i \mid d \mid \text{hash } exp \mid \text{sig } exp \; exp \mid \text{multisig } [exp] \; [exp] \mid \text{size } exp$$

Example 3. Considering informally the computation of the output script from Example 1, Table 1(left) shows the effect of command execution on SK. Dotted lines represent initial pointers.

Table 1. An example of symbolic stack (**left**) and the input types (ty^1, ty^2) and result type (ty^r) for some SCRIPT operators (**right**).

DUP;POP;POP;DUP	$[\ldots \; d_{-1}, \; (x_0, \top), \; (x_0, \top), \; \ldots]$ $f^{\wedge} \nearrow \quad \nwarrow^{\wedge} h$	
POP;POP;DUP	$[\ldots \; d_{-1}, \; d_0, \; d_1, \; \ldots]$ $f, h^{\wedge}$	
DUP	$[\ldots \; (x_{-1}, \top), \; (x_{-1}, \top), \; d_1 \; \ldots]$ $f \searrow \qquad h \searrow$	

OP	ty^1_{op}	ty^2_{op}	ty^r_{op}
ADD SUB	Int	Int	$bs_{[0..5]}$
LT LTE GT GTE	Int	Int	Bool
SIZE	$\top$	-	Int
EQ	$\top$	$\top$	Bool
AND OR	Bool	Bool	Bool
HASH256	$\top$	-	bs_{32}

5.2 Symbolic simulation of SCRIPT computations

The possible symbolic executions of an output script S are defined by a symbolic transition system, whose states represent the computation still to be executed and the current state of the associated symbolic state. Commands operate on typed data, type errors cause a *runtime error*, which stops the execution in a *failed* state, modelled here as standard as a non-terminal state (not $\diamond$) with no outgoing transitions. Table 1(right) reports examples of typed operations.

Definition 1. *A* symbolic state *for a* SCRIPT *program S is a tuple (S, SK, h, f), with SK a symbolic stack and h and f its head and floor indexes.*

Definition 2. *A* symbolic transition system *for a* SCRIPT *program S is a relationship between symbolic states, and a constraint store Γ, written as*

$$\Gamma \vdash (S_1, SK_1, h_1, f_1) \rightarrow (S_2, SK_2, h_2, f_2)$$

and read as Γ justifies the transition from (S_1, SK_1, h_1, f_1) to (S_2, SK_2, h_2, f_2).
Γ is a constraint store *over the variables $x_{f_2+1}, \ldots, x_{-1}, x_0$.*
$\rightarrow$ is the transition relation *amongst states.*
Both Γ and $\rightarrow$ are defined by the structural operation semantics *rules in Figure 2, 3 and 4.*

Γ may contain constraints like $\{(x_i, int), x_i \leq 100\}$: x_i is an integer variable whose value must be less than 100. We use juxtaposition of constraint stores for their union. For the sake of space we do not enter in the details of the definition of the constraint language and solver, as they are standard techniques over the domain of interest.

$\Gamma \vdash (S_1, SK_1, h_1, f_1) \rightarrow (S_2, SK_2, h_2, f_2)$ reads as *the program S with the stack (SK_1, h_1, f_1) can do a computation step, transform the stack into (SK_2, h_2, f_2), and become the program S_2, under the conditions in Γ*. The intended use of transactions is to define Γ through the semantic rules for a computation step of a given S_1 and (SK_1, h_1, f_1).

The union of the Γs along a computation made of several steps defines the *minimal* requirements on the initial stack to make that computation happen. Such a union for a *successful* execution trace defines one (or more) of the possible outputs of the *input script* that makes the transition redeemable. It is important to remark that one condition for success is that the top of the stack holds true. In order to validate such condition we

1. add a VERIFY operation at the end of the *output script* under consideration, which will cause the constraint $e_h = True$ to be added to Γ - see Figure 3, and
2. resolve successful termination as Γ satisfiability (and script's termination).

Definition 3. *Let SK_ϵ be the empty stack. A* successful trace *for a program S is a finite sequence*

$$\Gamma_0 \vdash (S, SK_\epsilon, 0, 0) \rightarrow (S_1, A_1) \ldots \Gamma_n \vdash (S_{n-1}, A_{n-1}) \rightarrow (\diamond, SK_n, h_n, f_n)$$

such that $\Gamma_0 \ldots \Gamma_{n-1}$ is satisfiable.

Γ satisfiability means that there exists an assignment γ such that $\Gamma\gamma$, i.e. the grounding of Γ through γ, is consistent. Interestingly, γ defines $x_{f+1}, \ldots, x_0$, the stack variables that have been identified by Γ and need to be to instantiated by the *input script*.

Theorem 1 (Soundness). *Let*

$$\Gamma_0 \vdash (S, SK_\epsilon, 0, 0) \rightarrow (S_1, A_1) \ldots \Gamma_n \vdash (S_{n-1}, A_{n-1}) \rightarrow (\diamond, SK_n, h_n, f_n)$$

be a successful trace for the script S, i.e. $\Gamma_0 \ldots \Gamma_{n-1}$ is satisfiable.

Then there exist a script I and an assignment γ such that $\Gamma\gamma$ is consistent, and the execution of I from the empty stack provides the stack $X\gamma = [x_{f+1}, \ldots, x_0]$ and the actual execution of $I; S$ is successful from the empty stack.

Proof. Sketch! (this proof relies on semantics rule described in the following). We prove the stronger result: if

$$\Gamma_0 \vdash (S, \mathbf{SK}, \mathbf{h}, \mathbf{f}) \rightarrow (S_1, SK_1, h_1, f_1) \ldots \Gamma_n \vdash (S_{n-1}, A_{n-1}) \rightarrow (\diamond, SK_n, h_n, f_n)$$

is a successful trace then exists I such that $I; S$ is successful starting from a ground instance of SK, according to Γ. By induction on the trace length n:

Case $n = 1$. S consists of the added VERIFY operation (S was initially empty), the only constraint is that I is able to provide a $x_h = True$ value on the stack, which define $I = $ OP_TRUE, and trivially $I; S$ is successful.

Case $n \Rightarrow n + 1$. For each single semantic rule r that can be applied to S, let us consider the step

$$\Gamma_0 \vdash (S, SK, h, f) \rightarrow (S', SK', h', f')$$

By construction a successful trace for S' of length $n-1$ exists (it goes to $\diamond$ and $\Gamma_1 \ldots \Gamma_n$ is satisfiable if Γ is), and by induction I' exists such that $I'; S'$ is successful. Depending on the rule r applied, it is possible to show that I exists, such that $I; S$ is successful.

Case $r = d - push$. We take SK' as the stack after the execution of PUSH d from (SK, h, f). By induction, since a successful trace exists for (S', SK', f', h') of length $n - 1$, then I' exists such that $I'; S'$ is successful from (S', SK', f', h'), with SK' equals to SK with on top the pushed datum d, and f'=f and h'=h+1.

It follows that $I'; $ PUSH d$; S'$ is successful from (SK_0, h, f), indeed we will have a suitable ground instance of SK (given that Γ is satisfiable) after the concrete computation of I', the execution of PUSH d will yield (a ground instance of) SK' from which we know that S' is successful.

Other cases can be solved analogously. □

$$\frac{h = f}{\{(X_f, \top)\} \vdash (SK, h, f) \xmapsto[\;(X_f, \top)\;]{pop} (SK, h-1, f-1)} \text{ s\_pop} \qquad \frac{h > f}{\emptyset \vdash (SK, h, f) \xmapsto[\;SK[h]\;]{pop} (SK, h-1, f)} \text{ pop}$$

$$\frac{}{\emptyset \vdash (SK, h, f) \xmapsto[\;(e,t)\;]{push} (SK[(e_{h+1}, t)], h+1, f)} \text{ push}$$

(a) Stack semantics rules

$$\frac{d\ is\_a\ t \quad \Gamma \vdash A \xmapsto[\;(d,t)\;]{push} B}{\Gamma \vdash (\text{PUSHDATA } d, A) \to (\diamond, B)} \text{ d-push} \qquad \frac{\Gamma \vdash A \xmapsto[\;(i, \text{BS}_1)\;]{push} B}{\Gamma \vdash (\text{OP\_}i, A) \to (\diamond, B)} \text{ op n}^5$$

$$\frac{\Gamma_1 \vdash A \xmapsto[\;(e,t)\;]{pop} B \quad \Gamma_2 \vdash B \xmapsto[\;(e,t)\;]{push} C \quad \Gamma_3 \vdash C \xmapsto[\;(e,t)\;]{push} D}{\Gamma_1 \Gamma_2 \Gamma_3 \vdash (\text{DUP}, A) \to (\diamond, D)} \text{ dup}$$

(b) Constants and stack ops

$$\frac{\Gamma_1 \vdash A \xmapsto[\;(e_1, t_1)\;]{pop} B \quad \Gamma_2 \vdash B \xmapsto[\;(e_2, t_2)\;]{pop} C \quad \begin{array}{c} ty_{op}^1 \triangleright t_1 \\ ty_{op}^2 \triangleright t_2 \end{array} \quad \Gamma_3 \vdash C \xmapsto[\;(e_1\ op_{AB}\ e_2, ty_{op}^r)\;]{push} D}{\Gamma_1 \Gamma_2 \Gamma_3 \vdash (op_{AB}, A) \to (\diamond, D)} \text{ 2-ops}$$

(c) Arithmetic & Boolean ops

Fig. 2. Semantic rules I

Rules in Figure 2a model auxiliary operations ($\xmapsto[\;X_f\;]{pop}$) transforming a symbolic stacks, hereafter ranged over by $A, B, C, \dots$. These operations are used by many of the other semantics rules and may define constraints for the constraint store. Worth noting the rule s_pop: data is expected (to have been provided by the input script) but the stack is empty: a new symbolic variable x_f is allocated in the first available position on the stack, i.e. f, and added to Γ with $\top$, i.e. no requirements, as type.

Rules for constants and stack ops are straightforwardly, examples are in Figure 5.2.

Arithmetic and boolean ops follow a common scheme: data are popped from the stack and, if types are correct, the result is pushed on the stack. Figure 2c shows the scheme for binary operations, most of which are defined in Table 1.

Flow control is described in Figure 3. Rule seq is the core of a small-step semantics: a sequence of commands is unfolded one at the time. ite_t checks the value popped from the stack: if it is, or can be assumed to be a $Bool$ according to type coercion rules, then ITE reduces to its if branch under the (minimal) assumptions that the type is a

---

[5] With $n \in [1..16]$

$$\frac{\Gamma \vdash (cmd, SK_1, h_1, f_1) \to (\diamond, SK_2, h_2, f_2)}{\Gamma \vdash (cmd; S, SK_1, h_1, f_1) \to (S, SK_2, h_2, f_2)} \ \text{seq}$$

$$\frac{\Gamma \vdash A \xmapsto[(e,t)]{pop} B \quad Bool \rhd t}{\Gamma\{t \ is \ Bool\}\{e = True\} \vdash \ (\text{IFE} \ S_t \circ S_f; S, \ A) \to (S_t; S, \ B)} \ \text{ite}_t$$

$$\frac{\Gamma \vdash A \xmapsto[(e,t)]{pop} B \quad Bool \rhd t}{\Gamma\{t \ is \ Bool\}\{e = False\} \vdash \ (\text{IFE} \ S_t \circ S_f; S, \ A) \to (S_f; S, \ B)} \ \text{ite}_f$$

$$\frac{\Gamma \vdash A \xmapsto[(e,t)]{pop} B \quad Bool \rhd t}{\Gamma\{t \ is \ Bool\}\{e = True\} \vdash \ (\text{VERIFY}, A) \to (\diamond, B)} \ \text{ver}$$

Fig. 3. Semantic rules II

$$\frac{\Gamma_1 \vdash A \xmapsto[(e,t)]{pop} B \quad \Gamma_2 \vdash B \xmapsto[(\text{hash}(e),BS_{32})]{push} C}{\Gamma_1\Gamma_2 \vdash \ (\text{HASH256}, A) \to (\diamond, C)} \ \text{h256}$$

$$\frac{\Gamma_1 \vdash A \xmapsto[d_{pk}]{pop} B \quad \Gamma_2 \vdash B \xmapsto[d_{sk}]{pop} C \quad \Gamma_3 \vdash C \xmapsto[(\text{Sig } d_{sk} \ d_{pk}, Bool)]{push} D}{\Gamma_1\Gamma_2\Gamma_3 \vdash \ (\text{CHKSIG}, A) \to (\diamond, D)} \ \text{chksig}$$

$$\frac{\begin{array}{ccc} \Gamma_1 \vdash A \xmapsto[n_{pk}]{pop} B & \Gamma_3 \vdash C \xmapsto[n_{sk}]{pop} D & \Gamma_5 \vdash E \xmapsto[-]{pop} F \\ \Gamma_2 \vdash B \xmapsto[pks]{pops \ n_{pk}} C & \Gamma_4 \vdash D \xmapsto[sks]{pops \ n_{sk}} E & \Gamma_6 \vdash F \xmapsto[(\text{MultiSig } sks \ pks, Bool)]{push} G \end{array}}{\Gamma_1\Gamma_2\Gamma_3\Gamma_4\Gamma_5\Gamma_6 \vdash \ (\text{CHKMSIG}, A) \to (\diamond, F)} \ \text{chkmsig}$$

Fig. 4. Semantic rules III

$Bool$ and the value is $True$. Note that, as a general rule, a type error prevents that application of the rule, not allowing termination and therefore a successful trace. ite_f follows straightforwardly, as well as the omitted rules for IF, and the (only) one for VERIFY - not $Bool$ or $false$ prevent termination.

The crypto ops are in Figure 4. $h256$ (and the omitted $h160$) describes the hashing of the top value in the stack. Similarly, $chksig$ pops a signature and a public key and pushes a symbolic expression of validating the (signature, public key, transaction message) pair. Analogously, $chkmsig$ pops a number n_{sk}, pops n_{sk} signatures, pops a number n_{pk}, pops n_{pk} public keys, pops 1 irrelevant value[6] and pushes a symbolic expression of validating the multiple signatures with multiple public keys to the stack.

5.3 Implementation

The presented symbolic verification framework has informed the implementation of SCRIPT ANALYSER, an open source application implemented in Haskell, available at `https://git.science.uu.nl/r.klomp/BitcoinAnalysis`.

Given an *output script* S, the current version of the tool returns all the existing satisfiable Γ for each successful computation of S. Such Γs are specifications of (all the possible) *input scripts* I which can be used to redeem the associated transaction. SCRIPT ANALYSER works by an exhaustive traversal of the space of successful traces, as soon as an error or inconsistency in Γ is detected, the trace is abandoned.

Satisfiability of Γ is done by application of well-known Finite Domain Constraint Solvers. The current version of the tool uses the solver embedded in *swi-prolog* [4]. Other solvers, e.g. *GNU Prolog* [2], will be experimented with.

An extensive experimentation over the non-standard transactions appeared in the blockchain is being carried out.

6 Two non-standard transactions

We present the analysis of two relatively complex *output scripts* from the blockchain. These scripts have been chosen as complex enough examples to make non-trivial the precise understanding of their intended meaning and the full conditions for redeemability. As such, any introduced bugs during development of SCRIPT programs like these would arguably be difficult to notice without formal verification.

```
1  OP_DUP;
2  OP_SIZE;
3  OP_PUSH (int: 64);
4  OP_PUSH (int: 67);
5  OP_WITHIN;
6  OP_SWAP;
7  OP_SHA256;
8  OP_PUSH <bs length: 32>;
9  OP_EQUAL;
10 OP_BOOLAND;
11 OP_IF;
12   OP_DROP;
13 OP_ELSE;
14   OP_PUSH <bs length: 65>;
15   OP_CHECKSIGVERIFY;
16 OP_ENDIF;
17 OP_PUSH <bs length: 65>;
18 OP_CHECKSIG
```

```
1  OP_IF;
2    OP_2;
3    OP_PUSH <bs length: 65>;
4    OP_PUSH <bs length: 33>;
5    OP_2;
6    OP_CHECKMULTISIGVERIFY;
7  OP_ELSE;
8    OP_2;
9    OP_PUSH <bs length: 65>;
10          % bs differs from bs at line 3
11   OP_PUSH <bs length: 33>;
12          % bs differs from bs at line 4
13   OP_2;
14   OP_CHECKMULTISIGVERIFY;
15 OP_ENDIF
```

(a) Example A (b) Example B

Fig. 5. Two output SCRIPT examples.

---

[6] This is conforming to the Bitcoin Core client, which contains a bug resulting in this additional stack entry to be popped from the stack.

Figure 5a shows the output SCRIPT of a transaction[7] that was inserted into the Blockchain's 269,760th block. Following the symbolic rules, there are two different Γs derivable:

$\Gamma_0 :=$ Sig X_{-1} $bs_{\text{line: 17}} \wedge$
  (
    Hash $X_0 == bs_{\text{line: 8}} \wedge$
    Size $X_0 < 67 \wedge$
    Size $X_0 >= 64$
) $\wedge$
  ...omitted type constraints...

$\Gamma_1 :=$ Sig X_{-1} $bs_{\text{line: 17}} \wedge$
  Sig X_0 $bs_{\text{line: 14}} \wedge$
  $\neg($
    Hash $X_0 == bs_{\text{line: 8}} \wedge$
    Size $X_0 < 67 \wedge$
    Size $X_0 >= 64$
) $\wedge$
  ...omitted type constraints...

The former (Γ_0) implies that the *true*-branch of the OP_IF instruction is taken, and the latter (Γ_1) implies that the *false*-branch is taken. For both Γs the input script must instantiate variables $\{X_0, X_{-1}\}$ and X_{-1} must be a valid signature. If the transaction is redeemed following Γ_0's constraints, X_0 must be a valid hash input such that the result is equal to some constant byte string and its type is constrained by the Size constraints to: $\{64,65,66\}$. Whereas, if the transaction is redeemed following Γ_1's constraints, X_0 must be a valid signature.

Figure 5b shows the output SCRIPT of a transaction[8] that was inserted into the Blockchain's 290,456th block. Again, since both branches of the OP_IF instruction produce valid constraint sets, the tool finds two solutions for Γ:

$\Gamma_0 := X_{-4} = true \wedge$
  MultiSig $[X_{-2}, X_{-1}][bs_{\text{line: 3}}, bs_{\text{line: 4}}] \wedge$
  $X_0 = true \wedge$
  $(X_{-3}, \top) \wedge$
  ...omitted type constraints...

$\Gamma_1 := X_{-4} = true \wedge$
  MultiSig $[X_{-2}, X_{-1}][bs_{\text{line: 9}}, bs_{\text{line: 11}}] \wedge$
  $X_0 = false \wedge$
  $(X_{-3}, \top) \wedge$
  ...omitted type constraints...

For both Γ_0 and Γ_1 variables $X_{-4}, .., X_0$ must be instantiated by the input SCRIPT. Depending on the value of X_0, the *true*-branch is taken (Γ_0, when $X_0 = true$) or the *false*-branch is taken (Γ_1, when $X_0 = false$). X_{-1} and X_{-2} must be valid signatures for both Γs. However, note that the public keys they must match with are different depending on which Γ is solved by the input SCRIPT. This shows that there exist *two* identity pairs that may redeem the transaction. X_{-3} is in both Γs popped from the stack due to the bug in MultiSig, and must only be present but is not further constrained. Additionally, in both Γs, X_{-4} must be some value equal to *true*. It can be argued that the inclusion of this constraint on variable X_{-4} is a minor bug of this output SCRIPT that is caused by the _VERIFY variant of MultiSig that is applied as the final operation (following both the *true*-, as well as the *false*-branch). Though the trained eye should have

---

[7] With ID: 75bb6417afc7500a6389201a67bfc2428a1241170a214bbf6833a389191036fe
[8] With ID: cd2dacbd05389580cb569985b3a8b1db67ea6cc84371223590e241a5026d0a8a

no trouble spotting this bug in the code, it is clear that the constraints generated by our prototype tool better highlight the bug's presence.

7 Conclusions

We introduced a symbolic analysis of *open* SCRIPTs. An *open* SCRIPT is an incomplete program, i.e. the *output scripts* in Bitcoin's transactions. These can be *closed* by prepending a set of instructions, i.e. the *input scripts* in Bitcoin's transactions. Through application of the symbolic evaluation rules, the constraints expressed by an *output script*, which are the ones that must be met by the *input script*, can be derived automatically, and be further analysed, either manually or (partially) automatically. We have shown that these constraints can, for example, show the non existence of a redeeming *input script*, e.g. due to type error(s) in the *output script*, or contradiction(s) in the constraints imposed by the *output script*. Results have been presented of analyses on two relatively complex non-standard *output scripts*. Such results have been obtained automatically by means of an open source application that we developed. Such results, beyond confirming that the two *output scripts* are redeemable, clarify by means of the generated constraints the required encrypted knowledge.

Currently, the evaluation rules and the prototype tool cover a relevant portion of SCRIPT's language. Interesting research for future work involves extending both, e.g. starting by the inclusion of locktime operations. On a longer term, we are planning to extend the symbolic evaluation to the analysis of smart contracts that are defined with multiple linked transactions in conjunction with off-chain communication, à la cryptographic protocol analysis.

Bibliography

[1] Github - bitcoin/bitcoin: Bitcoin core integration/staging tree. `https://github.com/bitcoin/bitcoin/`. Accessed: 12-06-2018.

[2] The gnu prolog web site. `http://gprolog.org/`. Accessed: 18-06-2018.

[3] Script - bitcoin wiki. `https://en.bitcoin.it/wiki/Script`.

[4] Swi-prolog. `http://www.swi-prolog.org/`. Accessed: 18-06-2018.

[5] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Modeling bitcoin contracts by timed automata. In *Int.al Conference on Formal Modeling and Analysis of Timed Systems*, pages 7–22. Springer, 2014.

[6] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 443–458. IEEE, 2014.

[7] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *International Conference on Financial Cryptography and Data Security*, pages 494–509. Springer, 2017.

[8] Massimo Bartoletti and Roberto Zunino. Bitml: a calculus for bitcoin smart contracts. Technical report, Cryptology ePrint Archive, Report 2018/122, 2018.

[9] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96. ACM, 2016.

[10] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 104–121. IEEE, 2015.

[11] Sergi Delgado-Segura, Cristina Pérez-Sola, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomartı. Analysis of the bitcoin utxo set. In *The 5th Workshop on Bitcoin and Blockchain Research*, 2018.

[12] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*, pages 79–94. Springer, 2016.

[13] David Gerard. Smart contracts, stupid humans: new major ethereum erc-20 token bugs batchoverflow and proxyoverflow. https://davidgerard.co.uk/blockchain/2018/04/26/smart-contracts-stupid-humans-new-major-erc-20-token-bugs-batchoverflow-and-proxyoverflow/, 2018.

[14] Stefano Lande and Roberto Zunino. Sok: Unraveling bitcoin smart contracts. In *Principles of Security and Trust: 7th International Conference, POST 2018, Held as Part of ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10804, page 217. Springer, 2018.

[15] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Bitcoin project white paper*, 2009.

[16] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[17] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.