A Formal Description of Bitcoin's Script Language

Domenic Cianfichi, Hugo Mailhot and Joseph McGee
Department of Computer Science
University of California, Davis
1 Shields Ave, Davis, CA 95616, USA

***Abstract*— Bitcoin transactions are validated using a specialized language called Script. Despite the critical role it plays in the Bitcoin network, so far no formal description of the language has been developed to allow reasoning about the expressive power of the language. The recent years have shown that a lack of understanding of Script made it possible to create valid programs with undesirable behavior. Furthermore, the Bitcoin community acknowledges that standard transaction script formats, and maybe Script itself, might be changed in the future. In a first step to allow rigorous reasoning about Script's expressivity and limitations, we present a formal description of the language, along with example analyses that illustrate the usefulness of such a description.**

I. INTRODUCTION

Bitcoin is a fully decentralized payment system, the first practical decentralized solution to "the double spending problem." Bitcoin has in the years since its introduction grown to prominence, with dozens if not hundreds of crypto-currencies following in its wake. Bitcoin is generally the simplest and most-established of these, serving as the anchor currency, against which other crypto-currencies are valued.

There has been a great deal of research into different aspects of Bitcoin. One particularly central aspect has, however, been under-examined. The transactions so secured by the Bitcoin network are encoded in Script, a programming language tasked with determining the validity of a transaction.

Script, as with the the entirety of the Bitcoin protocol, lacks a formal specification. Rather, it is defined by the grammar and serialization implied by the reference implementation of Bitcoin. We set out to provide a formal description of the Script language and to demonstrate, with somewhat simple examples, how such a description might be useful in analyzing the properties of the language.

Script is purposefully restricted, most notably lacking a mechanism for looping or recursion, guaranteeing termination. A a stack-based language written with byte literals as program text and no formal specification to turn to, Script is not an easy language to pick up and understand. It is a similarly difficult task to analyze it's behavior without a formal grammar.

In the past, Script has been progressively restricted in response to discovered vulnerabilities in the language. The restrictions eliminated such vulnerabilities at the cost of reducing the expressiveness of the language. It is not clear that all of the restrictions added were entirely necessary to secure the language.

This is complicated by the lack of any sufficiently precise definition of security. It is our hope that a formal description and eventually specification of Script will aid the construction of framework for such determinations, allowing more objective and precise analysis of both current and proposed future forms of the language.

To prevent attacks on the Bitcoin network based on exploiting a flaw in Script, the Bitcoin community agreed on a set of "standard" script templates that nodes will process. These limitations, however, are deemed temporary and subject to change in the future [3], although no consensus exists at the moment on what should replace these limitations, if anything. As Bitcoin's economic presence grows, formal specifications, abstracted away from current implementations of the Bitcoin protocol, will become increasingly essential in mediating conflict over what Bitcoin is or should be, allowing participants in the Bitcoin community to reason formally and precisely about their opinions and proposals.

II. APPROACH

A. Definition of Script Grammar

The bulk of our project consists in providing a well-formed and consistent definition of the Script language. This includes a description of the syntax that the text of scripts must conform to as well as descriptions of the operational semantics by which the scripts evaluate. Due to the breadth of this component of our work, we defer exposition of the grammar to [Appendix A].

As the Bitcoin protocol lacks any formal specification by which it is defined, it is well accepted that the reference implementation defines the protocol. We therefore turned to the reference implementation to construct our description of Script's grammar—while also heavily upon the [Bitcoin Wiki] and [Mastering Bitcoin] to gain a detailed if not authoritative conceptual understanding of the relevant concepts and constructs. Insomuch as our grammar is consistent with this reference, it is correct. Insomuch as it diverges it is incorrect. Formally proving the consistency or equivalence of our grammar with the reference implementation is outside the scope of this project and is left as future work. Indeed, our grammar is purposefully incomplete as discussed below (§II.C) due to the disproportionate expressive burden imposed by a few op-codes.

1) Syntax: Script is a stack-based language, bearing some semblance to Forth. It is parsed and evaluated left to right [Bitcoin Wiki/script]. The syntax of Script is defined by the binary serializations of script that reference implementation

understands to be well-formed. We discover this by reviewing relevant literature (of which there is not much) and the source code itself, to glean such syntactic structures.

The "text" of a Script program is its binary serialization. In most contexts, our syntax (accurately) considers this text to be tokenized into a byte vector. Ultimately all syntactic constructs in Script are either commands which alter the state of the program in a defined manner or contextual data which parameterize the behavior of some commands.

We divide commands into "simple-word commands", termed `scom`, which are single byte commands that take no contextual program text to modify their behavior; "multiple-word commands", termed `mcom`, which rely upon contextual program text to modify their behavior—such as `OP_IF`/`OP_ELSE`/`OP_ENDIF` or `OP_PUSHDATA1`; and disabled commands, termed `dcom`, which consist of op-codes reserved for future use, disabled to remove perceived attack vectors, or reserved for testing. Commands are constructed from op-codes or words which are single byte literals with special meaning along with any relevant contextual data.

We define our syntax sufficiently to be aware of the transactions formats in which scripts are embedded in the hope that it may be used in the future to describe the semantics of commands which require such awareness—`OP_CHECKSIG`.

2) Operational semantics: We provide both big-step and small-step operational semantics.

The state of the program consists entirely of the stack, S, the alt-stack, AS, and the validity V. The stack contains byte-vectors. The reference implementation defines the encoding used to map from byte-vectors to numbers. We take the existence of an inference rule describing said encoding for granted.

We describe the semantic meaning of each command with one or more corresponding inference rules. In order to contain the size of our description, we provide that well-formed programs with no corresponding inference rule for the execution of a command in a given state transition to the state in which the script is determined to be invalid and execution halts.

A number of op-codes have been disabled due to perceived vulnerabilities, reserved for future use, or reserved for internal use in testing. We provide inference rules for disabled op-codes, but also include them in `dcom`. We provide a contradictory inference rule for `dcom` which causes relevant commands to unequivocally evaluate to an invalid state. We provide these to allow for the modeling of the behavior of Script before these were disabled. If they are to be used, these op-codes must not be considered to be in `dcom`.

Scripts exist in transactions and op-codes exist which directly reference their transaction context. We omit these because of the disproportionate descriptive burden they introduce.

We also provide small step semantics to describe the progression of evaluation of a script program. The local and global reduction rules are compact due to the language's inherent simplicity.

B. Application of Script Grammar

[Describe how grammar was used]

C. Difficulties encountered

• The source code of a script program is just a sequence of bytes and is not human-readable. To make documentation easier, the creators of the language defined an alias (*opcode*) for each command. We first thought that the whole language could be described using only the opcodes, without referring to byte sequences as such, but the very semantics of certain commands made this impossible (see for example the `OP_PUSHDATA` commands in Section 2 of the appendix). It took us a few tries to come up with a notation precise enough to describe source code as byte sequences, and yet elegant enough to allow concise expression of the semantics.

• It was our hope that we could define operational semantics for op codes of previous versions of Script that caused crashes or other issues in the past. Using these definitions, we would then show that the old definitions could be used to cause problems. Unfortunately, there is very little documentation regarding how these errors came about. The reference documentation itself only documents changes back to 2014 while most of the serious problems with Script were fixed in 2010. The only fix we were able to analyze was the change to `OP_RETURN`.

• `OP_CHECKSIG` and related op-codes refer to the serialized form of the surrounding transaction. This is important as the signature which `OP_CHECKSIG` verifies serves dually to prove that the author of the transaction is in control of the relevant private key (corresponding to the public key which hashes to the address to which the relevant locked bitcoins were spent) and in fact authored the transaction presently under consideration. Due to the complexities of referencing the serialized form the transaction with big-step semantic inference rules, we did not, after much struggle, provide such rules for such op-codes.

III. APPLICATIONS AND RESULTS

In this section, we make use of our formal specification to reason about execution time bounds of Script programs and to explain how a flaw in a prior version of the language was fixed.

A. Programs written in the current version of Script halt within time linear in the length of the program

1) Background: For the Bitcoin network to function properly, it is important that the network form a single connected component. A type of attack against the network is to try and cut some nodes from the rest of the network, and then "double-spend" the same amount of money on the two disconnected components, which will both accept the transactions. Such a cut can be achieved by performing a DoS attack against a group of nodes that collectively ensure connectivity between two otherwise separated subnetworks. One way to perform a DoS attack could be to have a node attempt to evaluate a Script program with an unreasonable execution time. It is therefore important to guarantee that

a Script program will terminate within a certain amount of steps.

2) Analysis: Using the small-step semantics defined in Section 3 of the appendix, we see that no reduction rule yields its entire input, as was the case with the `while` local reduction rule in IMP. That is to say, no recursion is allowed. At most a proper subset of the input is conserved after applying the rule, when it is not completely replaced by `OP_NOP`, Script's equivalent of the `skip` command. This guarantees that every small-step reduction progresses towards termination of the program. Furthermore, the global reduction rule trivially entails that for any program that is not only `OP_NOP`, there exists a unique context reducible by a local reduction rule. Thus we have that every Script program halts.

We also know, from our description of Script's big-step semantics in Section 2 of the appendix, that every command requires a constant number of checks and operations on the stack. So we have that a Script program will always halt, and that the number of operations required to evaluate it is at most linear in the amount of commands it contains.

Given that the main nodes on the Bitcoin network enforce a limit on the length of Script programs one can include in a transaction, guaranteed linear execution time results in an absolute maximum number of steps for a standard Script program. This helps in protecting against DoS attacks based on maliciously crafted transactions. Furthermore, this maximum number of steps was deliberately lowered by requiring that the most computationally expensive opcode, `OP_CHECKSIG`, be contained exactly once in a valid standard transaction [4].

B. Previous versions of Script allowed skeleton key scripts

1) Background: The way an amount on Bitcoin is spent on the network is by evaluating to $\sigma[V = valid]$[1] the concatenation $C = BA$ of a locking script A, provided by the payer, and an unlocking script B provided by the recipient. The intended scenario is that A will contain a validation condition, which will check whether the output of B is adequate. Typically, B will have to provide a hash that proves they are the intended recipient of the money spent by the payer, and A will know what to expect as a proof. In that way, A is a guarantee to the payer that the money can only be received by the intended recipient. However, a careless definition of the `OP_RETURN` opcode in previous versions of Script made it possible to program skeleton key scripts that could spend any amount of money, without proving the recipient's identity.

2) Analysis: A skeleton script would have the following property:

$$\exists B.\forall A.\ BA \Downarrow \sigma[V = valid]$$

This is equivalent to saying that either (1) B is such that for any A, BA evaluates to a valid state, or (2) B evaluates to a state from which any A will evaluate to a valid state. Case (2) was never possible, since one could write a script $A =$ `OP_0`, that pushes 0 on the stack. Upon normal program termination, if the top stack value is 0, then the program is deemed invalid. Thus for this A there is no script B such that $BA \Downarrow \sigma[V = valid]$. Case (1), however, was made possible by the semantics of `OP_RETURN` prior to July 31 2010. In local reduction rule form, `OP_RETURN` was defined as follows:

$$< \texttt{OP\_RETURN com}, \sigma > \ \longrightarrow \ < \texttt{OP\_NOP}, \sigma >$$

Evaluating `OP_RETURN` results in the immediate termination of the program, *without any changes to the state*. Consider now a script $B =$ `OP_1 OP_RETURN`, and the evaluation of BA for any arbitrary script A:

1) `1` is pushed onto the stack
2) `OP_RETURN` triggers program termination
3) The top stack value is nonzero, and so the program is deemed valid

Clearly, that such a script was possible completely defeated the transaction validation process. `OP_RETURN` semantics were subsequently changed to the following:

$$< \texttt{OP\_RETURN com}, \sigma > \longrightarrow < \texttt{OP\_NOP}, \sigma[V = invalid] >$$

thus ensuring that its evaluation in a script invariably resulted in a rejected transaction. As no other command in the language can cause early termination of the program with a valid state, we can now say that there exists no skeleton key script that could serve to unlock any arbitrary locking script.

IV. RELATED WORK

To our knowledge, no previous effort was made to provide a formal description of Script. The Bitcoin developers community relies mostly on a combination of the Bitcoin wiki Script article [2] and Script's C++ reference implementation [1] to learn the language or reason about it.

The Bitcoin wiki Script article contains an explanation of how each opcode works in plain English, and information about the standard script templates that main nodes of the network will accept. While it does provide an informal explanation for each opcode (which came in handy when writing the operational semantics), its descriptions are rather unrigourous, and sometimes even ambiguous. For example, the explanation of `OP_LEFT` is that it takes an argument to index a string and throws away everything to the right of the index, but it does not specify whether that operation is right-inclusive. Finally, this resource cannot be used as input in automated theorem proving software.

The C++ reference implementation, by necessity, completely specifies the semantics of the language. However, the implementation also specifies details that are irrelevant to the syntactic correctness or valid/invalid judgement of a Script program. For example, the reference implementation defines a series of error codes, one for each possible cause of early termination with invalid state. These errors all cause the program execution to halt in the same way, and mainly serve to communicate the cause of an early termination to the running environment. Taking these various error codes

[1] See Appendix section 2 for a description of state configurations.

into account needlessly complexifies the type of analyses we presented in our results section. Additionally, as is the case with the existing Bitcoin wiki resource, we cannot directly use the language's implementation as axioms and rules to be used in a theorem prover.

Our proposed grammar, in constrast, unambiguously describes the behavior of Script, is easier to reason about the C++ code, and can (hopefully eventually) serve as basis for further analytic work.

V. CONCLUSIONS

REFERENCES

[1] Bitcoin Core. (March 2017). *Bitcoin Core* [Online; Accessed: 15-March-2017]. Available: https://github.com/bitcoin/bitcoin.

[2] Bitcoin Wiki. (March 2017). *Script* [Online; Accessed: 15-March-2017]. Available: https://en.bitcoin.it/wiki/Script.

[3] Andreas M. Antonopoulos. (October 18 2016). *Mastering Bitcoin - Unlocking digital currencies* [Online; Accessed: 15-March-2017]. Available: https://github.com/bitcoinbook/bitcoinbook.

[4] CVE Details. (March 3 2012). *Vulnerability Details : CVE-2013-2292* [Online; Accessed: 15-March-2017]. Available: https://www.cvedetails.com/cve-details.php?t=1&cve_id=CVE-2013-2292.