Chaussette: A Symbolic Verification of Bitcoin Scripts

Vincent Jacquot[1][0009−0007−8026−5277] and Benoit Donnet[1][0000−0002−0651−3398]⋆

Université de Liège, Montefiore Institute, Belgium
{vjacquot, benoit.donnet}@uliege.be

Abstract. The Bitcoin protocol relies on scripts written in Script, a simple Turing-incomplete stack-based language, for locking the money carried over the Bitcoin network. This paper explores the usage of symbolic execution for finding transactions that permit to redeem the money without being the legitimate owner. In particular, we show in detail how using insecure scripts could have led to security breaches, resulting in bitcoins theft. Our contributions include (*i*) a quantification of the vulnerable script instances over the full Bitcoin history up to Feburary, 4$^{\text{th}}$ 2023; (*ii*) the development and open source publication of a symbolic execution tool, called Chaussette; (*iii*) the description of how to use Chaussette to perform the attack; and, (*iv*) a discussion around a way to secure vulnerable money.

1 Introduction

Bitcoin, the first decentralized cryptocurrency has been deployed in 2008 [42]. While numerous cryptocurrencies followed thereafter [26,12], Bitcoin is the largest by market cap: half a trillion USD as of April 2023 [28]. Since then, the development of cryptocurrencies has generated a popular enthusiasm. In particular, for Bitcoin, multiple and various use cases have been implemented both by the academic and developers community [7] [5, Chapter 7]. Among others, we can cite lotteries [9,39], multiparty computations [3,11], or contingent payments [8,38].

In parallel to this enthusiasm, cryptocurrencies have been the subject, throughout the years, of several hijacking [35,33]. According to TRM Labs analysis, 2022 was a record-setting year for crypto hacks, with about $3.7 billion in stolen funds, including 10 hacks involving $100 million or more [47].

One key aspect of Bitcoin is that all applications share the common property of being handled by scripts written in the Script programming language. Indeed, Bitcoin relies fully on scripts to check the ownership and the validity of money expenses. These scripts are subject to bugs or vulnerabilities [13], introducing the risk of getting hacked and losing money.

In this paper, we provide a security analysis performed on the whole Bitcoin *blockchain* using Chaussette, a symbolic execution tool we developed. Chaussette explores all the paths a script's execution might take and searches for a

⋆ This work is supported by the CyberExcellence project funded by the Walloon Region, under number 2110186.

set of input values that allow a money transfer. In addition, we publicly release the Chaussette code.[1]

Our analysis shows that numerous scripts, more than three hundred thousand, do not properly secure the money they are in charge of. In particular, these insecure scripts allow people other than the true owner to spend the money. In total, tens of bitcoins could have been stolen.

The remainder of this paper is organized as follows: Sec. 2 provides a comprehensive guide on the main building blocks of the Bitcoin protocol and Script; Sec. 3 introduces Chaussette, with the results of the analysis on the whole blockchain. The attacks we found are carefully described and quantified; Sec. 5 discusses a potential solution to secure vulnerable funds; Sec. 6 positions this paper with respect to the state of the art; finally, Sec. 7 concludes this paper by summarizing its main achievements.

2 Background

In this section, we provide the required background for the remainder of the paper. In Sec. 2.1, we discuss the main concepts on which the Bitcoin protocol [42] (hereafter abbreviated as Btc), is built. In Sec. 2.2, we provide a comprehensive guide on the mechanisms used to verify the ownership of the currency defined by the protocol: the bitcoin (hereafter abbreviated as ₿).

2.1 The Bitcoin Protocol

The Bitcoin protocol (Btc) defines a decentralized digital currency that enables payments to anyone, anywhere in the world. The *satoshi* is the smallest possible division and equals one hundred millionth of a bitcoin (₿).

Btc runs over a decentralized network of nodes running a consensus algorithm for updating a public ledger of financial transactions. Those transactions are grouped into blocks which are chained together and form the *blockchain*.

Transactions are verified and blocks are created by special nodes called *miners*. In every new block, a given number of new ₿ is created and attributed to the miner as a reward. Initially, the reward was set to 50 ₿ and this value is halved every 210,000 blocks [5, Chapter 10]. Additionally, every transaction specifies a transaction fee paid to the miner which includes the transaction in the block.

Btc is designed to produce a block every ten minutes on average [5, Chapter 10]. The transactions that are broadcasted on the network wait in the *mempool* for a block to be mined.

A transaction is composed of a set of $n > 0$ inputs: $i_0, \ldots, i_{n-1}$ and $m > 0$ outputs: $o_0, \ldots, o_{m-1}$. The rightmost transaction represented in Fig. 1 is composed of two inputs and two outputs. Every output is defined by a script that locks the money and its value in satoshis. Here, the two outputs respectively lock 10^7 and $9,096,749$ satoshis. Every input refers to an output from a previous transaction and contains a proof of ownership. From that point, the referred

[1] See `https://gitlab.uliege.be/bitcoin/symbolic_execution`

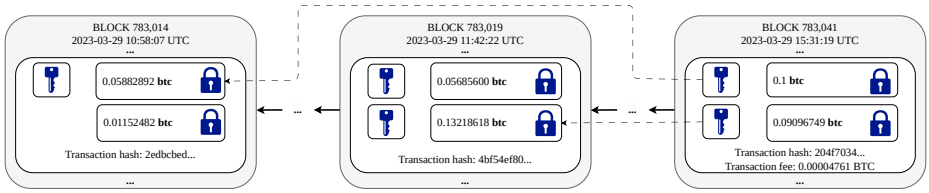


Fig. 1: The satoshis unlocked by the two inputs in the right-most transaction are split between two outputs. The transaction fee is attributed to the miner who includes the transaction into the block.

outputs' values are considered spent. Thus, the miners will reject any further transaction containing an input spending one of them. The two inputs claim respectively $5,882,892$ and $13,218,618$ satoshis from previous outputs. The sum of the values unlocked by the n inputs of a transaction has to be greater or equal to the sum of the values of the m outputs. The difference of the two sums (in the example: $4,761$ satoshis) is the transaction fee that is rewarded to the miner including the transaction into the block.

2.2 The Bitcoin Script Language

BTC defines a stack-based language, SCRIPT, to determine whether an input is allowed to spend or not an output. This language instructions are encoded over one byte and support a wide range of general functionalities such as cryptographic, arithmetic, or branching operations [19]. Some other operations allow pushing byte vectors onto the stack. When used as numbers, byte vectors are interpreted as little-endian variable-length integers with the most significant bit determining the integer sign [19].

Every input and output contains a script, which are both concatenated and executed by the miners. If no error occurs and the script returns `True`, the input is allowed to spend the money. Any non-zero value is interpreted as *True*, but its default representation is the byte vector $0x01$ [20]. On the other hand, `False` is represented by any representation of 0, such as an empty byte vector (its default representation [20]), $0x00$, or $0x80$ (negative zero). BTC specifies a set of standard scripts [14] that are well known and secure methods to lock an output. While the use of standard scripts is recommended, users can implement their own, i.e., non-standard, scripts to support their specific needs.

Fig. 2 illustrates the process of validation for an output locked with a standard PUBKEY script. By extension, the output and the corresponding input are said to be of type PUBKEY.

Firstly, the miner extracts the input and output scripts, that are provided in hexadecimal format in Fig. 2. Then, the scripts are parsed and concatenated. The parsing is straightforward: the first opcode 48 (72 in decimal) in the input script indicates the following 72 bytes stand for a constant. This completes the parsing of the input script. The output script starts with the opcode 41 that indicates a

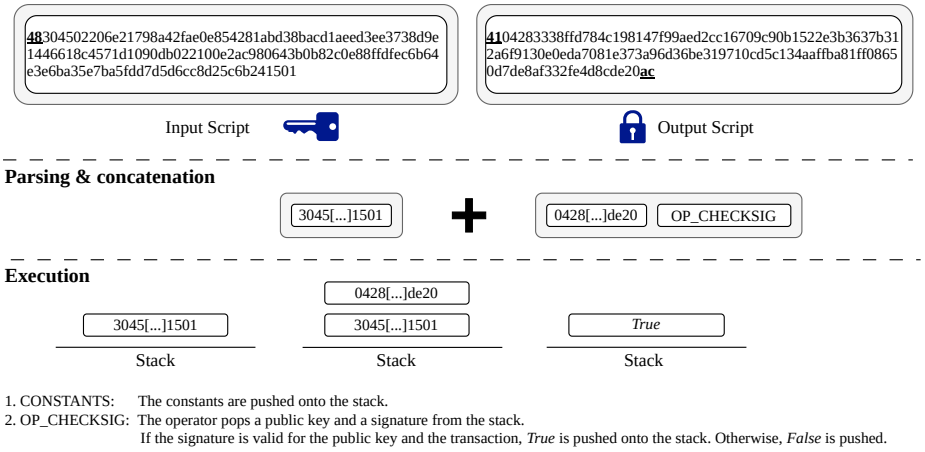


Fig. 2: Claim of a PUBKEY output.

constant of 65 bytes is following. Finally, the last opcode is *ac* standing for the operator OP_CHECKSIG [19].

The miner's last step is to execute the concatenation of the input and output script. The constants are pushed on the stack in LIFO order. OP_CHECKSIG pops two elements from the stack. The first one is assumed to be a public key and the second one a signature. A hash digest is obtained from the transaction. The exact parts of the transaction that are considered to produce the hash [16] are not discussed in this paper. The signature used by OP_CHECKSIG must be a valid signature for this hash and public key. If it is, *True* is pushed onto the stack. Otherwise, *False* is pushed onto the stack.

We also need to cover two other standard scripts defined by BTC: SCRIPTHASH [2] and WITNESS_V0_SCRIPTHASH [37] which require an extra verification rule.

Every SCRIPTHASH and WITNESS_V0_SCRIPTHASH input contains a second script called the *redeem script* that is included as the last constant inside the input script. For example, in Fig. 3, the constant $5121022afc[...]52ae$ is the redeem script. For the sake of simplicity, we do not present the raw hex script, but rather its parsed version. Note that this example stands for one particular instance. Redeem scripts are not restricted to the use of a standard MULTISIG script. In fact, redeem scripts might also be non-standard.

As usually, the miners will execute the input and output script together as illustrated at step 1 in Fig. 3. The redeem script is just interpreted as a constant. Then, a few extra steps are required for the transaction to be valid. The redeem script is parsed again. In the current example, the first byte is 51 that represents the instruction OP_1 and the second byte indicates the presence of a 33-byte constant: $022afc[...]$. To finish, 52 and ae stand for the instructions OP_2 and OP_CHECKMULTISIG. Finally, the parsed redeem script is executed with the

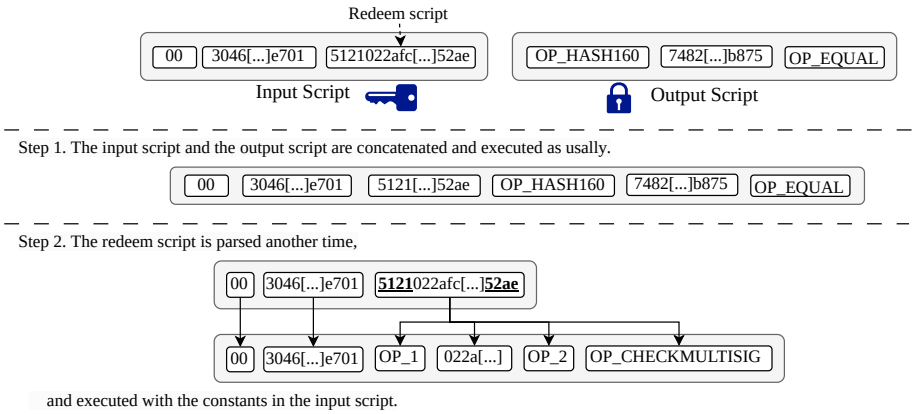


Fig. 3: Claim of a SCRIPTHASH output whose redeem script is a standard MULTISIG script.

remaining input data. This second execution must also run without errors and return *True* for the input to be valid.

3 Data Collection Methodology

Measurements were run on a computer equipped with an AMD 3600X processor running Ubuntu 20.04.5 at 4.4 GHz using 16 GB. We ran a BTC node on the machine to obtain the full blockchain. The client used was the C++ reference implementation [18] version v22.0.0. This client offers a convenient command line interface to fetch the transactions in JSON format. The blockchain was analyzed from block 0 to 775,000 (included). This latter was published on February 4$^{\text{th}}$, 2023 at 13:14:22 UTC.

Unfortunately, the command line interface does not include much information about the transactions' inputs in the results as they just point to an output [17]. In fact, the type and value of the corresponding output are not included. To circumvent the issue, instead of relying on existing pieces of code, such as Blocksci [32] (no longer supportered by its authors as of November 2020 [31]) we developed our own tool to parse the blockchain and annotate the inputs (our code is freely available[2]). The annotation is composed of two phases.

During the first phase, the blockchain is parsed exactly once. Unspent outputs are collected and cached in a UTXO (Unspent Transaction Outputs) set in RAM. Because of the RAM constraints, this set can only contain a maximum number of outputs. When the set is full, the oldest unspent outputs are evicted from the cache. The inputs are annotated with their corresponding output, if this latter is in the set.

[2] `https://gitlab.uliege.be/bitcoin/symbolic_execution`

The second phase consists in labeling the inputs that were not annotated during the first phase. The RAM is filled with as many of these inputs as possible. Then, the blockchain is parsed to find their corresponding outputs. This procedure is repeated with another batch of unannotated inputs until all inputs are annotated.

Our code performed the two phases in roughly 48 hours. Our code will be released upon paper acceptance.

4 Non-Standard Scripts as Attack Vector on Bitcoin

In this section, we expose how non-standard scripts can be used as an attack vector to steal funds. Those custom scripts (see Sec. 2.2) are implemented by the users or services to protect their funds. They can be involved either in inputs as a redeem script or in outputs. As with every piece of code, they are subject to bugs and vulnerabilities.

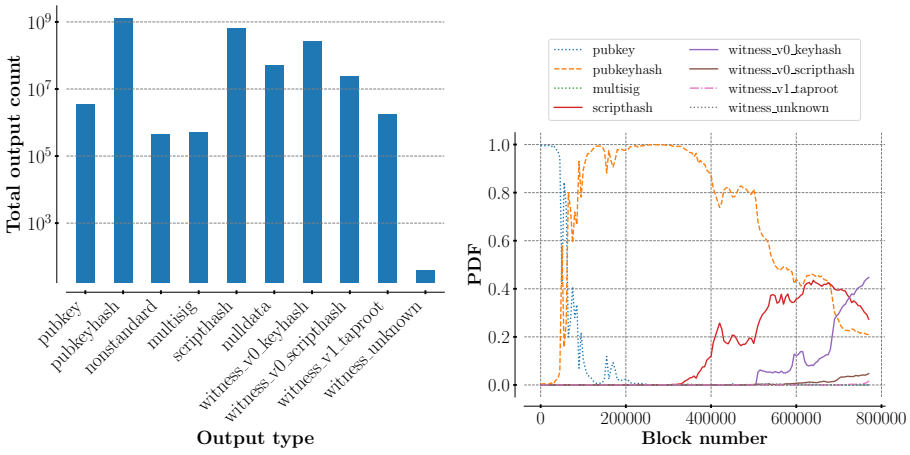
Fig. 4a exposes a few metrics to give an order of magnitude of the different scripts' usage. 433,458 outputs are locked with a non-standard script, which represents 0.019% of all the outputs. 24,394,307 WITNESS_V0_SCRIPTHASH [37] and 634,004,474 SCRIPTHASH [2] outputs have been used. From these 658.3 million outputs (29% of all outputs), 16.4M are unspent. Amongst the 641.9 million inputs spending the outputs, 3,435,086 `redeem scripts` (0.15% of all outputs) are non-standard. This gives a total of 3,868,544 non-standard scripts found in the blockchain.

Finally, the probability distribution function (PDF) per output type is provided in Fig. 4b. Most of the variations in usage come from the fact that all the standard scripts have been defined at different points in time. For example, SCRIPTHASH was defined in January 2012 [2] (roughly after block 160,000) and took a long time to be widely adopted.

While still popular, the usage of outputs requiring a redeem script decreases over time for the benefit of simpler locking methods such as PUBKEYHASH and WITNESS_V0_KEYHASH scripts.

While the usage of non-standard scripts stays low in proportion, this still concerns numerous outputs. More importantly, the majority of ₿ is held by a small number of Virtual Asset Service Providers (VASP), because many BTC users rely on them to manage their cryptocurrency [27]. A malicious VASP could start using on purpose a vulnerable non-standard script to protect the funds of their customers. Then, this could be used as a back door to steal the money and the VASP could claim to be under attack.

In the following subsections, two attacks on non-standard scripts are described. The tool implemented to perform these attacks is described priorly at Sec. 4.1. The first attack targets the non-standard output scripts (Sec. 4.2) and the second one targets the non-standard redeem scripts (Sec. 4.3).



(a) Total output types count (b) Output types share over the time

Fig. 4: Distribution of the BTC outputs up to block 775,000.

Table 1: Path and opcodes count of the non-standard scripts.

	Mean	Std Dev	95% CI
Path Count	5.622	2.87	±0.003
Opcodes Count	15.35	10.47	±0.01

4.1 CHAUSSETTE: A SCRIPT Symbolic Execution Tool

Symbolic execution is a way of analyzing a program to determine what inputs cause each part of a program to execute. Symbolic execution employs satisfiability-modulo theory (SMT) [40] constraint solvers to determine the feasibility of a path condition and generate concrete solutions for it.

Contrarily to miners who execute scripts on concrete values to check the validity, our symbolic execution tool, called CHAUSSETTE, executes scripts on symbols and returns scripts' output as a function in terms of these symbolic inputs. Then, CHAUSSETTE uses Z3 [41], an SMT solver, to find a set of concrete values for which the script output is `True`. For example, while a miner would execute the concatenation of an input script and an output script, CHAUSSETTE only considers the output script and searches for values for which the output script returns *True*.

This technique is particularly suitable for the Bitcoin script language as many of the usual limitations do not apply. Indeed, the scripts are usually quite simple (see Table 1), so the number of feasible paths stays computationally manageable. Additionally, the Bitcoin language does not implement arrays [19] that are usually trickier to represent symbolically [43]. Finally, no operator interacts with their environment as a regular program on a machine would, e.g., by making system calls or receiving signals [19].

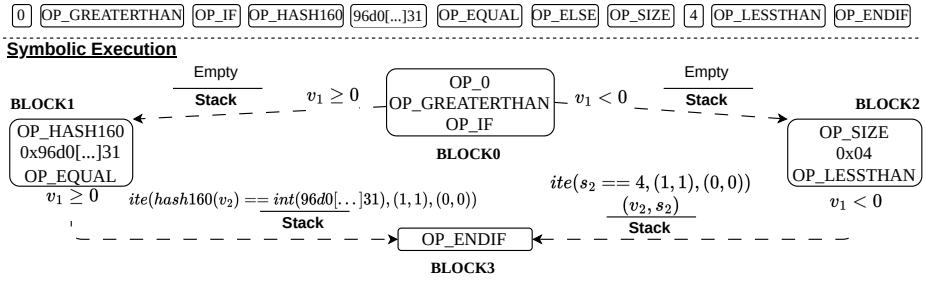


Fig. 5: Example of CHAUSSETTE execution.

Due to space constraints, we do not describe how CHAUSSETTE deals with every operator. Instead, we present how CHAUSSETTE deals with the script illustrated in Fig. 5. CHAUSSETTE's goal is to find the data that will redeem the money locked by this script. We use the acronym *ite* standing for `If-Then-Else`, such that the expression $ite(a, b, c)$ is evaluated to b if a is *True*, c otherwise. Finally, byte vectors used by SCRIPT are represented as tuples (v, s), where v is the integer interpretation of the vector and s its size in bytes. The default representation of *False* is the empty byte vector $(val = 0, size = 0)$, while the default representation of *True* is (1, 1) (see Sec. 2.2).

As a first step, CHAUSSETTE builds the control flow graph for the script. The opcodes are grouped into blocks of consecutive instructions, these blocks are chained together to represent all the possible paths. The full list of opcodes that might alter the execution flow is [OP-IF, OP-NOTIF, OP-RETURN, OP-VERIFY] [19]. These two latter are special in the sense that they may invalidate the execution. To represent this feature in our model, we create a special final block ERROR-BLOCK that pushes 0 (*False*) onto the stack and does not point to any block. Thus, whenever this block is visited, the top stack is necessarily *False* and invalidates the execution accordingly. The informed reader will notice that some branching operators are missing from the above list, such as OP_CHECKSIGVERIFY, OP_EQUALVERIFY, etc. The reason is that these operators ending with "verify" can be replaced by two operators. For example, the operator OP_CHECKSIGVERIFY can be replaced with the operators OP_CHECKSIG OP_VERIFY without any logic change [19].

The script is decomposed into four blocks: BLOCK0 containing the operators [OP-0 OP-GREATERTHAN OP-IF] that points to BLOCK1 and to BLOCK2 respectively composed of [OP-HASH160 96d0[...]31 OP-EQUAL] and [OP-ELSE OP-SIZE 4 OP-LESSTHAN]. These two blocks point to the final block of the script: BLOCK4 composed of [OP-ENDIF].

The second step is the symbolic execution. We cover in detail in the next paragraphs the execution for blocks $0, 1, 2$, and 3. Fig. 5 illustrates for each block the set of constraints and the state of the stack after execution which are passed to the child blocks.

BLOCK0 The first block to be executed is the entry point of the script. Firstly, OP-0 pushes the empty vector $(0, 0)$ onto the stack. Then, OP-GREATERTHAN tries to pop two elements from the top of the stack. As only one element is on the stack, CHAUSSETTE will generate on the fly a symbolic BTC byte vector. This symbolic vector is represented with a tuple (v_1, s_1), where v_1 stands for its integer value interpretation and s_1 is the number of bytes. Finally, OP-GREATERTHAN pushes another symbolic vector onto the stack represented with the tuple: $ite(v_1 \geq 0, (1, 1), (0, 0))$. Finally, OP-IF pops this symbolic vector. Contrarily to a regular execution, both child blocks, BLOCK1 and BLOCK2, will be executed with a different set of constraints: respectively $v_1 \geq 0$ and $v_1 < 0$. Additionally, the state of the stack, that is currently empty, is passed to the children.

BLOCK1 As OP-HASH160 tries to pop an element from an empty stack, a symbolic vector (v_2, s_2) is generated. OP-HASH160 pushes the 20-byte vector [19] $(hash160(v_2), 20)$ onto the stack, where $hash160$ is an uninterpreted function taking one integer as an argument and returning one integer. As a remainder, an uninterpreted function is a function that has no other property than its name and a n-ary form. It allows any interpretation that is consistent with the constraints over the function. Then, the constant $96d0[...]31$ is pushed onto the stack. OP-EQUAL pushes the vector $ite(hash160(v_2) == int(96d0[...]31), (1, 1), (0, 0))$. Finally, the set of constraints and the state of the memory are passed to BLOCK3.

BLOCK2 A symbolic vector (v_2, s_2) is generated as OP-SIZE needs an element and the stack is empty. OP-SIZE pushes onto the stack the element itself (v_2, s_2), then the size of this element $(s2, min\_bytes\_encoding(s2))$ where $min\_bytes\_encoding$ is a function returning the size needed to encode $s2$. Then, the concrete byte vector $(4, 1)$ is pushed onto the stack. Finally, OP-LESSTHAN pops two elements from the stack and pushes $ite(s2 < 4, (1, 1), (0, 0))$ on it. No additional constraint is added to the path. As done for BLOCK1, the state of the memory and the set of constraints are passed to BLOCK3.

BLOCK3 As this block has several parents, we need to reconcile both paths' constraints and memories. The parents' constraints are merged using a logical OR. This gives us the expression $v_1 \geq 0$ OR $v_1 < 0$, which simplifies into *True*, which is expected as any execution of the script goes through that block. The memories are reconciled using *ite* expressions. For example, *top*, the top stack element, becomes:

$$top = ite(v_1 \geq 0,\ ite(hash160(v_2) == int(96d0[...]31), (1, 1), (0, 0))\\ ite(s_2 == 4, (1, 1), (0, 0)))$$

Table 2: Patterns for which the votes were not unanimous. The 3rd column designates the number of scripts for which CHAUSSETTE found a solution to unlock them.

Pattern	Script count	Vulnerable script count
OP-CONSTANT OP-CHECKLOCKTIMEVERIFYOP-DROP	15	7
OP-CONSTANT OP-CONSTANT	2	1
CONSTANT CONSTANT CONSTANT OP-CHECKMULTISIG	18	3

The last step consists in using Z3 to find values such as $top \neq 0$. Because of the nature of uninterpreted functions, $hash160$ is not constrained and Z3 can evaluate $hash160(v_2) == int(96d0[...]31)$ to *True*. This is equivalent to assuming we have the ability to perform a preimage attack, which is infeasible [36] for RIPEMD-160 and SHA256, the two hash functions being used by SCRIPT [19]. The trick is to add a final constraint $hash160(v_2) \neq int(96d0[...]31)$. Thus, Z3 will return a solution made of concrete values such as $v_1 < 0 \; and \; s_2 == 4$. In order to redeem the money locked by this script, two constants in the input script are required: the first one being a constant of 4 bytes, the second one being a negative value of any size.

We conclude this subsection with the results of CHAUSSETTE on the 3,868,544 non-standard scripts found. Despite the quantity, most scripts are very similar and can be grouped into 780 patterns. Two scripts containing the same opcodes, but differing only in the constants are said to be generated from the same pattern. In order to speed up the computation, up to 100 scripts (some patterns have fewer than 100 script instances) from every pattern have been randomly selected to be analyzed by CHAUSSETTE with a 30-second timeout. The final security tag attributed to the pattern is voted by majority and propagated to all the script instances generated from this pattern. A pattern can either be considered unsafe if a solution allowing the script to be unlocked is found, safe otherwise.

Apart from three patterns (see Table 2), the votes per pattern were unanimous. These three instances correspond to cases where the constant plays a predominant role in the semantic. For example, in the third pattern, the first constant designates the number of valid signatures the owner must present [19]. In three cases, this constant is 0, making the script vulnerable.

In total, CHAUSSETTE ran 16,138 script over 14,549.65 seconds (∼4h). Aggregating the run time per pattern, we obtain a 95% confidence interval for the run time per pattern of $1.33_{\pm 0.2448}$ seconds.

Moreover, due to the small number of patterns, we have inspected every one of them manually to assert the security tag correctness. We define as a positive a pattern tagged unsafe and a safe pattern as negative. Table 3 contains all the analysis results. Nine patterns over the 780 exceed the 30-second timeout, with five being manually analyzed as unsafe and four as safe. Two patterns use the only opcode that CHAUSSETTE does not support: OP-ROLL, one being safe and the other one unsafe. In total, eleven patterns and the 2,114 related scripts

Table 3: Results of the CHAUSSETTE analysis over the 780 patterns.

		Manual security tag		**Total**
		Positive	Negative	
CHAUSSETTE security tag	Positive	176	3	**179**
	Negative	2	588	**590**
	Timeout	5	4	**9**
	Not supported	1	1	**2**
Total		**184**	**596**	**780**

cannot be analyzed by CHAUSSETTE. Therefore, CHAUSSETTE is able to analyze 98.59% of the patterns discovered and 99.94% of the scripts that relate to these patterns.

Over the 771 patterns CHAUSSETTE manages to analyze within 30 seconds, we detected three false positives (FP) and two false negatives (FN) for 176 true positives (TP) and 588 true negatives(TN). Considering only the patterns that CHAUSSETTE is able to analyze within 30 seconds, the recall of CHAUSSETTE is 0.9888 and its precision is 0.9832.

The origin of false positives and false negatives can be traced back to our model that does not always perfectly align with reality. In general, a false positive arises when the model lacks certain constraints, leading it to be overly permissive in its formulation of a solution.

To ease reproducibility and future improvement of the current state of the art, CHAUSSETTE is publicly released upon paper acceptance.[3] Additionally, access to the analyzed scripts will be granted on demand to researchers.

4.2 Non-Standard Output Scripts

To perform this attack, the attacker needs to keep up-to-date a real-time UTXO (Unspent Transaction Outputs) set. In April 2023, it is composed of roughly 88M outputs [25] which is manageable for any decent computer. For every unspent output, CHAUSSETTE will return whether this script can be unlocked and the values of the constants to include in the input script.

Note that CHAUSSETTE does not generate the input script by itself, but this could be very easily implemented to fully automate the attack.

This attack can even be upgraded to a replay attack. Let us suppose an output o requires finding a value y such that its $sha256$ hash is x and does not involve any signature verification. As stated in the previous section, CHAUSSETTE is designed to assume that preimage attacks are impossible. However, in the very specific context of BTC, the legit owner must publish the input containing this value y in a transaction t. This transaction t is broadcasted and is waiting in the mempool to be included into the blockchain. One could sniff the mempool very easily, as every full node maintains one. Moreover, most clients propose a

[3] See `https://gitlab.uliege.be/bitcoin/symbolic_execution`

Table 4: Non-standard output scripts: security analysis results.

	Script count	Pattern count	Total value locked
Vulnerable to symbolic execution	220,554	18	1.946 75 ₿
Vulnerable to replay attacks	62	11	2.349 736 ₿
Safe	203,485	39	3962.236 896 ₿
Unspendable	9,357	16	0.639 538 ₿

Table 5: Non-standard input scripts: security analysis results.

	Script count	Pattern count	Total value locked
Vulnerable	153, 310	194	51.07 ₿
Safe	3,281,776	514	2.192923×10^6 ₿

very convenient way to fetch this data [15]. Then, one would publish another transaction t' claiming the same output o', but proposing a higher transaction fee to incentive miners to include t' rather than t into the blockchain.

The results in Table 4 summarize the attack's severity. The 433,458 scripts can be grouped into 84 distinct patterns. From the Sec. 4.1, it has been shown that eighteen of them are unsafe, and the 1.947 ₿ protected by the 220,554 scripts derived from them can be unlocked by anyone. On the other hand, the upgraded version of the attack could have been used to steal 2.35 ₿ from 62 scripts. In total, from the 10.63 vulnerable, 9.62 ₿ have been spent, leaving 1.01 ₿ vulnerable. Finally, 9,357 outputs have been proven to be impossible to spend and result from either a malformed script or a script designed to be unspendable [1]. This category also encompasses scripts whose execution was invalidated because of reserved opcodes [19].

4.3 Non-Standard Redeem Scripts

This second attack involves sniffing the mempool as for the previous replay attack in order to find inputs spending SCRIPTHASH and WITNESS_V0_SCRIPTHASH outputs. As a reminder from Sec. 2.2, these outputs include a value x and the miners check that the hash of the redeem script equals to this x.

The attacker only needs to parse the mempool to find a transaction t spending one SCRIPTHASH or WITNESS_V0_SCRIPTHASH output. If the redeem script does not involve any signature operators, the attacker can just publish a transaction t' with the same input, a larger transaction fee, and a different output to steal the money. Because miners tend to include transactions that maximize their profit, t' is more likely to be included than t [5, Chapter 2].

Table 5 contains the results of the analysis on the 3,435,086 non-standard redeem scripts we found. 51.07 ₿ have been spent with an input that could have been attacked, i.e., the redeem script does not involve any signature. While, this only represents an insignificant percentage (0.0023289%) of the total money that has flown through these outputs, it still represents a decent incentive for hackers.

There are exactly 16.4M unspent SCRIPTHASH and WITNESS_V0_SCRIPTHASH outputs, and they lock a total value of 5.46M ₿. This value is provided by TRM Labs[4], as it was faster to ask them rather than set up a UTXO set. This represents the only third-party data we used in this paper. Thus, it represents 26% of the 21M ₿ that will ever be mined [5, Chapter 1]. Unless they are including an x value already encountered in the blockchain, the corresponding redeem script is unknown. Thus, it is unfortunately impossible to know for every one of them if they are safe or not.

But, the presence of 153,310 vulnerable scripts over the 641.9M spent SCRIPTHASH and WITNESS_V0_SCRIPTHASH outputs, gives a vulnerability ratio of 0.000238838. Assuming the same ratio holds for the 16.4M unspent outputs, we are able to provide an estimation of 1,304 ₿ that might be stolen.

5 Attempts to secure Btc

This section starts with recommendations to secure the future published scripts against the attacks we have just described (Sec. 5.1). Then, a discussion on how to secure scripts that are already published is proposed (Sec. 5.2).

5.1 Recommendations

At first, companies and individuals should consider if they really need to use non-standard scripts because innovation introduces new risks as demonstrated in Sec. 4. In case the need is real, the use of tools such as Chaussette is necessary to assert the security.

As a rule of thumb, script developers should ensure that every possible path involves at least one signature operation. Moreover, they should keep in mind that relying only on pre-image hashes is not sufficient to guarantee security. Indeed, a transaction containing a secret is going to be made public before being included in the blockchain.

5.2 Securing Published Scripts

Unfortunately, there is no perfect solution for the published scripts that are currently vulnerable. However, this section explores a potential solution.

Let us consider the following context. Bob owns some ₿ in a SCRIPTHASH output. To spend it, Bob needs to publish the redeem script in an input in a transaction t and suppose this script is vulnerable as described in Sec. 4. From that moment, this transaction t is broadcasted, but not yet included in the blockchain as it is waiting for a miner to include it in a block. As far as we know, there is no mechanism implemented to prevent an attacker (Alice) performing a replay attack as described in Sec. 4.

Such a situation has already been discussed in 2013 by the Btc community as they faced the same issue [45]. A few outputs (see 37k7toV1Nv4DfmQbmZ8K

[4] `https://www.trmlabs.com/`

uZDQCYK9x5KpzP) were designed to be awarded to the first person finding a collision for $sha1$. The redeem script did not involve any signatures, thus making an attack possible.

By applying the same solution suggested in the thread, Bob would not broadcast t, but instead he would mine a block by itself and include t in it. The redeem script would become public at the same time as the output will be spent. Attacking this vulnerable output would require rewriting BTC history which is computationally impossible unless Alice owns 51% of the computation power [5, Chapter 10].

Unfortunately, this solution is impractical nowadays as the amount of computing resources to mine a block is greatly higher than in 2013 [24]. As an alternative, Bob could reach out to a known ₿ mining company [6] and provide them with proof of ownership. The mining company would not broadcast t and would include the transaction moving Bob's fund to a secured address directly in a block. The downside of this solution is that it relies on the ability to trust the mining company. From the moment this company is in possession of t, they could simply modify it to steal the money.

6 Related Work

Blockchain security has attracted the attention of the research community those last years. Numerous tools, such as Mythril [29], Securify [48], Manticore [46], and Oyente [30] have been developed to analyze and report security issues in Ethereum smart contracts. These tools also employ some symbolic analysis of the code.

To the best of our knowledge, no tool designed for BTC exists. A prototype tool [34] has been developed, but it only covers a portion of SCRIPT language and was tested on two real BTC scripts.

Bartoletti and Zunino [10] define a theory of liquidity and a verification technique for contracts expressed in BitML, a high level DSL (Domain Specific Language) for smart contracts that compile into BTC transactions.

Finally, Andrychowicz et al. [4] discuss a framework for modeling the BTC contracts using timed automata. They provide two BTC contracts that are modeled manually as an example. Unfortunately, no automation process has been provided yet.

7 Conclusion

While BTC is considered safe by most people, this paper highlighted vulnerabilities in non-standard scripts written in SCRIPT that could have led to the theft of 55.36 ₿, 1.57M US dollars worth on April, 26$^{\text{th}}$ 2023 [23]. A comprehensive guide on SCRIPT and its role in securing ₿ and the detailed attacks was provided. Moreover, a proposal to secure the ₿ that might still be vulnerable has been given.

The Btc blockchain was parsed up to block 775,000 and 3,868,544 non-standard scripts have been found. These scripts can be grouped into 780 patterns.

Chaussette, our symbolic execution tool, is capable of analyzing 99.94% of the Btc scripts within 30 seconds with a precision and a recall of respectively 0.9832 and 0.9888. Moreover, Chaussette highlights the presence of numerous insecure patterns used to secure ₿.

Potential future works include, but are not limited to, the application of Chaussette to assert other utxo blockchains' security which are also using Script [21,22] or the application of symbolic execution techniques to account-based blockchains.

Ethical Considerations

The researches discussed in this paper have been conducted in accordance to ethical considerations in blockchain network measurements [44]. Further, to avoid any security issue, the vulnerable scripts are not released publicly. Finally, Chaussette must be seen as a tool for also assessing vulnerability risks in using non-standard scripts.

References

1. Andresen, G.: Bitcoin core release notes 0.9.0, `https://github.com/bitcoin/bitcoin/blob/master/doc/release-notes/release-notes-0.9.0.md`, last Accessed: 20.04.2023
2. Andresen, G.: Pay to script hash. BIP 16, Bitcoin (January 2012)
3. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Fair two-party computations via bitcoin deposits. In: Proc. Financial Cryptography and Data Security (FC) (March 2014)
4. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Modeling bitcoin contracts by timed automata. In: Proc. Formal Modeling and Analysis of Timed Systems (FORMATS) (September 2014)
5. Antonopoulos, A.: Mastering Bitcoin. O'Reilly Media, Inc. (2014)
6. Arrieche, A., Henn, P.: Who are the biggest bitcoin mining companies?, `https://capital.com/biggest-global-crypto-bitcoin-mining-companies-ranking-btc#:~:text=What%20are%20the%20famous%20bitcoin,according%20to%20data%20from%20CompaniesMarketCap.`, last Accessed: 08.05.2023
7. Atzei, N., Bartoletti, M., Cimoli, T., Lande, S., Zunino, R.: Sok: Unraveling bitcoin smart contracts. In: Proc. Principles of Security and Trust (POST) (April 2018)
8. Banasik, W., Dziembowski, S., Malinowski, D.: Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In: Proc. European Symposium on Research in Computer Security (ESORICS) (September 2016)
9. Bartoletti, M., Zunino, R.: Constant-deposit multiparty lotteries on bitcoin. In: Proc. Financial Cryptography and Data Security (FC) (April 2017)
10. Bartoletti, M., Zunino, R.: Verifying liquidity of bitcoin contracts. In: Proc. Principles of Security and Trust (POSRT) (April 2019)

11. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: Proc. Advances in Cryptology (CRYPTO) (August 2014)
12. Binance: Binance Coin Whitepaper, https://www.exodus.com/assets/docs/binance-coin-whitepaper.pdf, last Accessed: 02.05.2023
13. Bistarelli, S., Mercanti, I., Santini, F.: An analysis of non-standard bitcoin transactions. In: Proc. Crypto Valley Conference on Blockchain Technology (CVCBT) (June 2018)
14. Bitcoin Community: Bitcoin improvement proposals, https://github.com/bitcoin/bips, last Accessed: 30.03.2023
15. Bitcoin Community: getrawmempool - bitcoin, https://developer.bitcoin.org/reference/rpc/getrawmempool.html, last Accessed: 01.06.2023
16. Bitcoin Community: Op checksig, https://en.bitcoin.it/wiki/OP_CHECKSIG, last Accessed: 07.08.2023
17. Bitcoin Community: RPC API reference, https://developer.bitcoin.org/reference/rpc/, last Accessed: 11.04.2023
18. Bitcoin Community: Running a full node, https://bitcoin.org/en/full-node, last Accessed: 30.03.2023
19. Bitcoin Community: Script, https://en.bitcoin.it/wiki/Script, last Accessed: 30.03.2023
20. Bitcoin Core developers: Bitcoin Core - interpreter.cpp, https://github.com/bitcoin/bitcoin/blob/80f4979322b574be29c684b2e106804432420ebf/src/script/interpreter.cpp#L412, last Accessed: 28.04.2023
21. Bitcoin Core developers: Dogecoin - script.cpp, https://github.com/dogecoin/dogecoin/blob/master/src/script/script.cpp, last Accessed: 02.05.2023
22. Bitcoin Core developers: Litecoin - script.cpp, https://github.com/litecoin-project/litecoin/blob/master/src/script/script.cpp, last Accessed: 02.05.2023
23. blockchain.com: Bitcoin price, https://www.blockchain.com/explorer/assets/btc, last Accessed: 26.04.2023
24. blockchain.com: Total hash rate (TH/s), https://www.blockchain.com/fr/explorer/charts/hash-rate, last Accessed: 26.04.2023
25. blockchain.com: Unspent transaction outputs, https://www.blockchain.com/fr/explorer/charts/utxo-count, last Accessed: 18.04.2023
26. Buterin, V.: Ethereum Whitepaper, https://ethereum.org/en/whitepaper/, last Accessed: 02.05.2023
27. Chainalysis: 60 % of bitcoin is held long term as digital gold. what about the rest?, https://blog.chainalysis.com/reports/bitcoin-market-data-exchanges-trading/, last Accessed: 13.04.2023
28. CoinMarketCap: Coinmarketcap, https://coinmarketcap.com/fr/, last Accessed: 26.04.2023
29. ConsenSys: Mythril, https://github.com/ConsenSys/mythril, last Accessed: 12.04.2023
30. Enzyme Finance: Oyente, https://github.com/enzymefinance/oyente, last Accessed: 12.04.2023
31. Kalodner, H., Möser, M., Lee, K., Goldfeder, S., Plattner, M., Chator, A., Narayanan, A.: Blocksci, https://github.com/citp/BlockSci, last Accessed: 07.08.2023
32. Kalodner, H., Möser, M., Lee, K., Goldfeder, S., Plattner, M., Chator, A., Narayanan, A.: BlockSci: Design and applications of a blockchain analysis platform. In: Proc. USENIX Security Symposium (August 2020)

33. Kessler, S.: Axie infinity's ronin blockchain overhauls tech, expands to new game studios a year after $625m hack. https://www.coindesk.com/tech/2023/03/30/axie-infinitys-ronin-blockchain-overhauls-tech-expands-to-new-ip-on-anniversary-of-600m-hack/, last Accessed: 02.05.2023
34. Klomp, R., Bracciali, A.: On symbolic verification of bitcoin's script language. In: Proc. Data Privacy Management, Cryptocurrencies and Blockchain Technology (DPM) (September 2018)
35. Korn, J.: Another crypto bridge attack: Nomad loses $190 million in chaotic hack, `https://edition.cnn.com/2022/08/03/tech/crypto-bridge-hack-nomad/index.html`, last Accessed: 02.05.2023
36. Li, Y., Liu, F., Wang, G.: New records in collision attacks on RIPEMD-160 and SHA-256. In: Proc. International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) (April 2023)
37. Lombrozo, E., Lau, J., Wuille, P.: Segregated witness (consensus layer). BIP 141, Bitcoin (December 2015)
38. Maxwell, G.: The first successful zero-knowledge contingent payment, `https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/`, last Accessed: 12.04.2023
39. Miller, A.K., Bentov, I.: Zero-collateral lotteries in bitcoin and ethereum. Proc. IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) (April 2016)
40. Monniaux, D.: A survey of satisfiability modulo theory. In: Proc. Computer Algebra in Scientific Computing (SASC) (September 2016)
41. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (March–April 2008)
42. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), `http://www.bitcoin.org/bitcoin.pdf`
43. Perry, D., Mattavelli, A., Zhang, X., Cadar, C.: Accelerating array constraints in symbolic execution. In: Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) (July 2017)
44. Tang, Y., Li, K., Wang, Y., Chen, J.: Ethical challenges in blockchain network measurement research. In: Proc. Workshop on Ethics in Computer Security (EthiCS) (February 2023)
45. Todd, P.: Topic: REWARD offered for hash collisions for SHA1, SHA256, RIPEMD160 and other, `https://bitcointalk.org/index.php?topic=293382.0`, last Accessed: 26.04.2023
46. Trail of Bits: Manticore, `https://github.com/trailofbits/manticore`, last Accessed: 12.04.2023
47. TRM Labs: Looking back at 2022 and towards 2023 to see what the future holds for digital assets policy. https://www.trmlabs.com/post/looking-back-at-2022-and-towards-2023-to-see-what-the-future-holds-for-digital-assets-policy (December 2022), last Accessed: 26.04.2023
48. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.: Securify: Practical security analysis of smart contracts. In: Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS) (October 2018)