# BSHUNTER: Detecting and Tracing Defects of Bitcoin Scripts

Peilin Zheng
Sun Yat-sen University
zhengpl3@mail2.sysu.edu.cn

Xiapu Luo
The Hong Kong Polytechnic University
csxluo@comp.polyu.edu.hk

Zibin Zheng*
Sun Yat-sen University
zhzibin@mail.sysu.edu.cn

***Abstract*—Supporting the most popular cryptocurrency, the Bitcoin platform allows its transactions to be programmable via its scripts. Defects in Bitcoin scripts will make users lose their bitcoins. However, there are few studies on the defects of Bitcoin scripts. In this paper, we conduct the first systematic investigation on the defects of Bitcoin scripts through three steps, including defect definition, defect detection, and exploitation tracing. First, we define six typical defects of scripts in Bitcoin history, namely unbinded-txid, simple-key, useless-sig, uncertain-sig, impossible-key, and never-true. Three are inspired by the community, and three are new from us. Second, we develop a tool to discover Bitcoin scripts with any of typical defects based on symbolic execution and enhanced by historical exact scripts. By analyzing all Bitcoin transactions from Oct. 2009 to Aug. 2022, we find that 383,544 transaction outputs are paid to the Bitcoin scripts with defects. The total amount of them is 3,115.43 BTC, which is around 60 million dollars at present. Third, in order to trace the exploitation of the defects, we instrument the Bitcoin VM to record the traces of the real-world spending transactions of the buggy scripts. We find that 84,130 output scripts are exploited. The implementation and non-harmful datasets are released.**

***Index Terms*—bitcoin, blockchain, smart contract**

## I. Introduction

Supporting the most popular cryptocurrency, the Bitcoin platform provides peer-to-peer payment in a decentralized way [36]. Its transactions for spending and receiving bitcoins are programmable via its scripts written in the bitcoin transaction script language, named Script [9]. Script, a stack-based language without loops, is not Turing-complete.

However, like other programs, the Bitcoin scripts may have defects that can lead to bitcoin loss. For example, once a user receives some bitcoins from the other, they will be locked in the transaction's output script. That is, the program controls the bitcoins. If the output script is buggy, attackers may take control of this payment by exploiting the output script's defects. Actually, there are already many such attacks. For example, [43] identified 884 bitcoin addresses with simple private keys, which are worth around $100K, and 21 of them were drained. We define it as the simple-key defect in this paper.

Although many studies on Bitcoin have been done, in view of money transferring [34], [35], [23], the security in the consensus and transaction [44], [40], and the software reliability [24], [45], only a few studies examine Bitcoin scripts with a focus on the quantitative statistics of non-standard scripts [10], stored content [32], and symbolic verification [29]. Unfortunately, to the best of our knowledge, none of the existing studies has investigated the defects of Bitcoin scripts. In this paper, we conduct the first systematic investigation on the defects of Bitcoin scripts. It is non-trivial to detect all buggy Bitcoin scripts due to the following challenges.

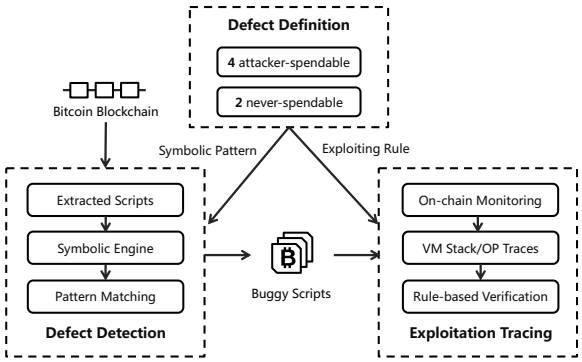



Fig. 1. Architecture of BSHunter

**C1: Lack of systematic defect definitions for Bitcoin scripts.** To the best of our knowledge, there is no systematic study on the defects in Bitcoin scripts, and the security of Bitcoin scripts has not received much attention. Moreover, defining a defect requires an in-depth study of various common patterns in Bitcoin scripts. It should also be considered with the identity attributes and economic motives behind their code patterns. Therefore, the definition of the defects is still blank. To fill the gap, we need to first clearly define the defects and then design a detection approach accordingly.

**C2: Lack of automated tools for detecting defects in Bitcoin scripts.** Bitcoin scripts can be divided into standard ones and non-standard ones. Bitcoin community defines seven standard script types (aka patterns) [21]. The scripts that do not match the patterns are considered non-standard. For the standard ones, we can use regular expressions to learn and check execution semantics when spending it. The execution semantics of the non-standard ones are diverse due to the infinite number of possible combinations of operation codes (opcodes). Moreover, as for the P2SH/P2WSH [21] scripts, their output scripts only show the hash values of the exact scripts. This makes it difficult to detect defects in the P2SH/P2WSH scripts. As shown in the statistics in August

TABLE I
DETECTED BUGGY BITCOIN SCRIPTS AS OF AUG. 2022.
(#TXOUT IS THE COUNT OF TRANSACTION OUTPUT SCRIPTS)

| | Defect | # TxOut | Bitcoin |
|---|---|---|---|
| **Attacker-spendable** | Unbinded-txid | 77,141 | 55.67 |
| | Simple-key | 1,484 | 21.89 |
| | Useless-sig | 38,284 | 0.21 |
| | Uncertain-sig | 43 | 0.25 |
| **Never-spendable** | Impossible-key | 205,320 | 3,002.28 |
| | Never-true | 61,272 | 35.12 |

2022 [8], P2SH scripts take up 45% of all scripts in one month. Hence, a tool for automatically checking all Bitcoin scripts, especially for P2SH/P2WSH/non-standard ones, is highly desired.

**C3: Lack of verification and tracing of the defects.** Once we detect the defects in the Bitcoin scripts, we find it challenging to prove the correctness of the detection since there is no ground truth. And it is also difficult to learn or prove whether the defects are exploited. Hence the method of verification and tracing is needed.

To detect and trace the defects of Bitcoin scripts and address the above challenges, we propose BSHUNTER, which consists of three modules, including defect definition, defect detection, and exploitation tracing, as shown in Figure 1.

**Defect Definition.** Since Bitcoin is designed for payment, we regard the issues in Bitcoin scripts that make the owners lose their control of payment as defects. Moreover, the defects could be attacker-spendable or never-spendable. After inspecting Bitcoin scripts, being inspired by the studies [10], [43] and community examples [17], we define four attacker-spendable defects (unbinded-txid, simple-key, useless-sig, and uncertain-sig) and two never-spendable defects (impossible-key and never-true), solving **C1**. Three of them (simple-key, unbinded-txid, and never-true) are inspired by the community, and the other three (impossible-key, useless-sig, uncertain-sig) are new from us. We also provide real-world examples for ease of explanation. For each defect, we propose the detection patterns in terms of symbolic expressions (§IV-D), which is the basis for automated detection as we intend to use the rule to match various scripts with different semantics. The details of defects are given in §III, and the threat to the validity of the definition is discussed in §VII.

**Defect Detection.** We propose a defect detection approach for Bitcoin scripts, solving **C2**. At first, we propose an address-based method as a naive solution. It uses the address interfaces of Bitcoin clients to detect partial defects from standard scripts. However, since the actual semantics of non-standard scripts are unknown, we develop a symbolic virtual machine for Bitcoin scripts, which follows the execution model in Bitcoin [17]. It begins with an initial state, gives symbolic inputs to a Bitcoin script, and operates its state according to the opcodes in the Bitcoin script. Once it faces conditional operations (e.g., If-Else), it will generate the branches according to different conditions. Then, it uses the Z3 SMT solver [20] to confirm whether the symbols can meet the conditions to spend the bitcoins (will be detailed in §IV-C). In this way, we can explore all the possible branches and states of a Bitcoin script. Moreover, as for the P2SH and P2WSH scripts, we conduct a historical-transaction-based database to find the exact scripts before execution. Finally, we define the symbolic patterns of defects (§IV-D) to match the final symbolic states (e.g., the used keys, the symbolic variables of signature). If any patterns match the states, then the script is considered to contain the defects corresponding to the patterns.

**Exploitation Tracing.** BSHUNTER instruments a Bitcoin full node to trace the real-world spending transactions of the buggy scripts, solving **C3**. Towards each transaction, we trace the operations and stacks for each step of execution in the Bitcoin VM. After that, we use several exploiting rules to check whether the defects are exploited. Note that the exploiting rules are based on actual execution and concrete input in the real world, unlike the symbolic values during detection.

**Result.** We apply BSHUNTER to 2,005,704,690 Bitcoin scripts from the first 749,000 blocks in Bitcoin, which contains all the output scripts from October 2009 to August 2022. We find that 383,544 output scripts in Bitcoin are with defects. Our proposed method can increase the number of detected scripts by 43% and the amount by 592% compared to the address-based method. The detailed results are shown in Table I. Among these scripts, 116,952 scripts are attacker-spendable, and 266,592 scripts are never-spendable. In total, the detected buggy Bitcoin scripts are making users lose 3,115.43 BTC, around 60 million dollars at present. The largest part of the increment (2609.36 BTC) comes from the 23 impossible-key scripts, which will be discussed in §VII. Moreover, by tracing the relevant transactions, we find that 99.6% bitcoins of the attacker-spendable scripts have been exploited. Note that anonymity in Bitcoin means that we cannot determine if exploits are actually by attackers or not. We release the implementation and the non-harmful data to prevent other users from losing their money[1].

**Contribution.** We conduct the first systematic study on defects of Bitcoin scripts with the following contributions.

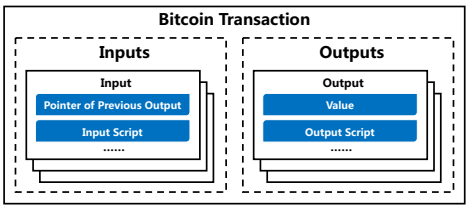
- We define six types of defects of Bitcoin scripts: unbinded-txid, simple-key, useless-sig, uncertain-sig, impossible-key, and never-true, with real-world examples.
- We design a defect detection approach based on symbolic execution and enhanced by historical exact scripts.
- We propose an exploitation tracing method to verify and trace the real-world exploitation of the defects.
- We apply our tool to real-world Bitcoin scripts and find 383,544 buggy output scripts where most attacker-spendable bitcoins have been exploited.

## II. BACKGROUND

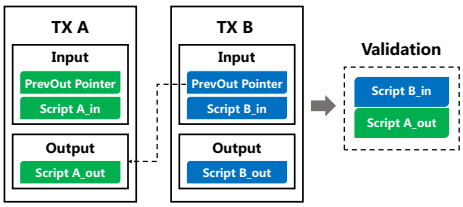
### A. Bitcoin Transaction

**Transaction Structure.** Figure 2(a) shows the structure of Bitcoin transactions. Each transaction consists of two lists:

[1]https://UnsafeBTC.com




(a) Structure of Bitcoin Transaction




(b) Bitcoin Transaction Validation Based on Scripts

Fig. 2. Bitcoin Transaction and Script

the input list (inputs) and the output list (outputs). Each input includes a pointer and a script. The pointer refers to a previous output. Each output includes a value of Bitcoin and a script. Note that the scripts in this paper are called input scripts and output scripts for ease of understanding. In some references [10], they are also called unlocking and locking scripts, respectively.

**Transaction Validation.** In Bitcoin, despite the coinbase input that issues bitcoin, each input needs to be validated whether it can spend the previous output. The validation is based on the execution of scripts, which will be detailed in the next subsection. If and only if all the inputs are valid, the transaction is successful and recorded into the blockchain. Note that Bitcoin transactions are all anonymous. Therefore, in the rest of this paper, once we find defects exploited in transactions, we cannot confirm whether the exploits are by attackers or owners.

### *B. Bitcoin Script*

Figure 2(b) shows the validation of a Bitcoin transaction, which is based on Bitcoin scripts. To simplify the introduction, we assume that both transactions (TX A and B) have only one input and one output. We also omit the output value. In this figure, TX A has been confirmed, and TX B is being validated whether it has the authority to spend the output of TX A.

**Input Validation.** The validation of input in TX B consists of 3 steps: (1) The blockchain client first checks out the output through the PrevOut Pointer (Pointer of Previous Output). If the output is an unspent transaction output (UTXO), the client will read its script for the next step. Actually, the bitcoin clients (C++/Golang/etc.) maintain a database of Unspent Transaction (TX) Outputs. Once the output in TX A is spent by TX B, the database will delete A:0 (the No.0 output in TX A) and then insert B:0 (the No.0 output in TX B). (2) The UTXO's script will be linked with B's input script, and then the linked script will be executed in the Bitcoin script engine. (3) If and only if the script execution has no failure and returns True on the top of the stack, the input is valid. Otherwise, the input is invalid.

TABLE II
EXAMPLE OF BITCOIN SCRIPT EXECUTION

| Operation | Stack After Execution |
|---|---|
| PUSH [signature] | [signature] |
| PUSH [pubkey] | [signature],[pubkey] |
| DUP | [signature],[pubkey],[pubkey] |
| HASH160 | [signature],[pubkey],[pubkey-hash] |
| PUSH [hash] | [signature],[pubkey],[pubkey-hash],[hash] |
| EQUALVERIFY | [signature],[pubkey] |
| CHECKSIG | True |

**Example.** To help to understand, an example of a Bitcoin script is introduced as follows. The input script is "PUSH [signature] PUSH [pubkey]", and the output script is "DUP HASH160 PUSH [hash] EQUALVERIFY CHECKSIG". The script execution is shown in Table II. After executing the operation code in order, the top of the stack is True, and then the input is valid.

**Script Type.** Commonly used Bitcoin scripts have the same patterns. Bitcoin community [21] defines eight types of Bitcoin scripts: (1) Pay-to-Public-Key: The output script records the public key. The input script is required to give the signature. (2) Pay-to-Public-Key-Hash: The output script records the hash value of the public key. The input script is required to give the public key and the signature. (3) Multiple-signature: The output script records N public keys. The input script is required to give K (less than N) signatures. (4) Pay-to-Script-Hash: The output script records the hash value of a raw script. The input script is required to give the corresponding raw script for validation. After validating the script hash, the script itself will be executed. (5) Pay-to-Witness-Public-Key-Hash: It is similar to the Pay-to-Public-Key-Hash. The difference is that some of the scripts are moved into a segregated witness structure in the transaction. (6) Pay-to-Witness-Script-Hash: It is similar to the Pay-to-Script-Hash, with the script in the segregated witness. (7) Null-data: The transaction output only records some data into the blockchain. This kind of script cannot be spent. (8) Non-standard: The scripts that do not match any of the above patterns are non-standard scripts.

For a Pay-to-Script-Hash or Pay-to-Witness-Script-Hash script, the actual executing script falls into two transactions, thus resulting in the challenge of defect detection. We detail it in §IV. Note that the Bitcoin community only provides the standard as advocacy rather than a limitation, protecting users' freedom, which has resulted in many non-standard scripts. Moreover, both standard and non-standard scripts could have defects introduced in §III.

## III. DEFECT DEFINITION

We collect and define the defects after examining the examples from the community (e.g., [10], [43], [17]) and discuss the threat to the definition and classification in §VII.

**General Definition.** Since the major function of the Bitcoin script is to give users control of the payment [36], we focus

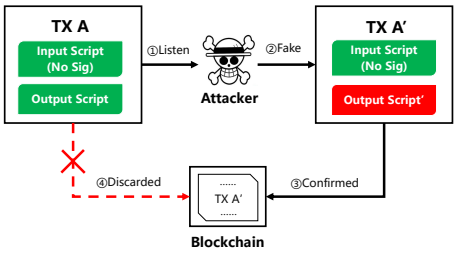



Fig. 3. Example of Unbinded-txid Scripts

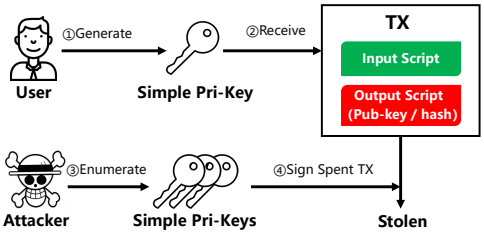



Fig. 4. Example of Simple-key Scripts

on the buggy Bitcoin scripts with the following characteristics: (1) Valuable: The script must carry non-zero bitcoins, which makes it valuable for the user. (2) User loses control: If the control of the spending script is not just owned by the user, or no one can spend it, it is regarded as buggy. Consequently, we divide the defects into two categories, namely attacker-spendable and never-spendable.

Since only the output script is used to lock the bitcoins, rather than the input script, we focus on the output scripts. The following subsections introduce four attacker-spendable defects (unbinded-txid, simple-key, useless-sig, and uncertain-sig) and two never-spendable defects (impossible-key and never-true), respectively. Note that the ideas of three defects (simple-key, unbinded-txid, never-true) are inspired by the references from the community [10], [43], [17], and the other three (impossible-key, useless-sig, uncertain-sig) are newly proposed by us. The differences are introduced at the end of this section.

### A. Unbinded-txid

**Definition.** In Bitcoin, "txid" is the hash value of the body of a Bitcoin transaction. The entire transaction's outputs, inputs, and scripts are hashed as $txid$. The $txid$ can be used by several operation codes (e.g., $CHECKSIG$, $CHECKMULTISIG$, $CHECKSIGVERIFY$, and $CHECKMULTISIGVERIFY$). For example, when executing $CHECKSIG$, the VM will call the signature and public key from the stack to decrypt them in order to check whether the decrypted result is the $txid$. In other words, once the script needs to verify the txid, then the output is binded to it, which will not be tampered with by an attacker. On the contrary, the transaction output has the risk of being tampered with. Hence, we regard such unbinded-txid scripts as attacker-spendable.

**Example.** As shown in Figure 3, before a transaction is confirmed in the blockchain, it will be propagated into the network. An attacker could listen to the network for the pending unbinded-txid transactions, then change the outputs to steal the Bitcoin. Once the attacker's transaction is confirmed, the original transaction will be discarded, losing the Bitcoin. For a real on-chain example, one transaction [1] pays 1 BTC to the script as "HASH256 PUSH 6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000 EQUAL", which can be spent by giving the pre-image of the hash value. Any attacker who listens to the network for the pre-image can change the output script to steal that 1 BTC.

### B. Simple-key

**Definition.** If the script requires the signature from a simple private key, it has a defect that is attacker-spendable. Previous studies [13], [12] have studied the impact of random number seeds on public-key security. On the contrary, in this paper, we focus on the users who do not use the random number seeds but generate simple private keys based on personal preferences, e.g., simple integers.

**Example.** Figure 4 shows an attacking example. The user generates a simple private key as he like, e.g., the integer 1. Then he uses the corresponding public key to conduct the output script to receive bitcoins. In the meanwhile, the attacker can enumerate several simple values to generate the simple private keys and the corresponding scripts. Once the attacker finds a transaction output of those scripts, he can sign a transaction to spend it, stealing the bitcoins. The on-chain examples will be introduced in §VI-C.

### C. Useless-sig

**Definition.** Bitcoin provides the operation code of signature verification for users. After the script execution of $CHECKSIG$ and $CHECKMULTISIG$, the verification result will be pushed back into the stack. However, if the verification result has no relevance to the final top of the stack, it is useless.

**Example.** As an on-chain example, one transaction [2] pays 0.0001 BTC to a Pay-to-Script-Hash script. And the exact executing script of it is "PUSH <pubkey> CHECKSIG NOT". Since the $CHECKSIG$ is followed by a $NOT$, its verification result is inverted. In this case, giving a wrong signature can unlock the bitcoins in that script. Hence, the $CHECKSIG$ is regarded as useless.

### D. Uncertain-sig

**Definition.** All Bitcoin operations of signature verification need public keys as parameters. In addition, the multi-signature verification will need the number of required signatures as a parameter. If the output script uses such operations without the parameters, then the script is with an uncertain-sig defect. It is attacker-spendable since anyone can input the parameters to steal the bitcoins.

**Example.** As an on-chain example, one transaction [3] pays 0.001 BTC to a Pay-to-Script-Hash script. However, the exact executing script of it is only "CHECKMULTISIG", without any numbers or parameters of public keys. In this case, any attackers giving the parameters can spend it.

### *E. Impossible-key*

**Definition.** Bitcoin has no restrictions on the public key in the script. Therefore, the user can use any public key to verify the signature in the script. However, not all public keys have matching private keys. Some public keys (e.g., all byte is 1) are impossible to have a private key. Scripts that require an impossible public key to verify will never be spent.

**Example.** For an real example, one transaction [4] pays 100 BTC to a non-standard script. The UTXO is "DUP HASH160 PUSH 0 EQUALVERIFY CHECKSIG". In this script, $CHECKSIG$ requires a public key, the hash of which must be 0. And this is the only way to pass the script execution. The current computers have to try infinite private-keys to get such "0" public-key-hash. The probability of generating such a 256-bit key is $\frac{1}{2^{256}}$, which is nearly impossible for current computers. Hence, the UTXO cannot be spent.

### *F. Never-true*

**Definition.** The script will never return "True" at the top of the stack that its Bitcoin will never be spent. Therefore, in this paper, all Nulldata scripts with non-zero amounts are considered vulnerable. However, some of them are actually intentional, for a protocol called Proof-of-Burn [46]. Moreover, a number of non-standard scripts are also never-true, which will be shown in §VI-C.

**Example.** As an on-chain example, one transaction [5] pays 0.0001 BTC to the a non-standard script. The script is "DUP HASH160 PUSH dc353b35d11614e1678d6607a2908f68eb76a007 EQUALVERIFY CHECKSIG PUSH 0". Because 0 is pushed in the end, the final stack top value of this script can only be 0, which is never-true.

Note that the never-true scripts are with 0% possibility to be spent, but the impossible-key scripts are with more than 0% possibility (if we can calculate out the private key, such as using the quantum computer in the future). Thus they are exclusive.

It is worth noting that three of the above defects are inspired by the community, not copied from the community. As for the simple-key, [43] has studied the key generated by brain-wallet (e.g., English Words), but we study the key as simple integers (e.g., 123). Thus the keys are exclusive between [43] and ours since they are from different perspectives. As for the unbinded-txid, [10] only shows the pattern of "OP_Hash160 OP_Equalverify", but [10] does not consider it unsafe. And our definition of unbinded-txid is also beyond this pattern in [10] (e.g., "OP_DUP OP_DROP" is also unbinded-txid). As for the never-true, [10], [17] only show the bitcoins with "OP_RETURN" are unspent, but our definition of never-true is beyond this in [10], [17] (e.g., $(x < 1$ && $x > 1)$ is also never-true). Hence, our three definitions are inspired by [10], [43], [17] but with obvious differences

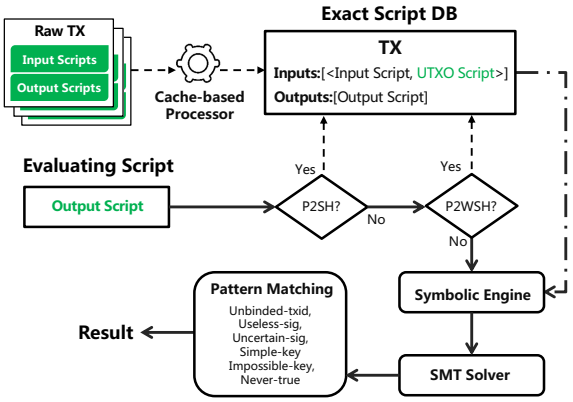



Fig. 5. Overview of Defect Detection

## IV. Defect Detection

After defining six defects of Bitcoin scripts, we will first propose a naive solution to partially detect them, and then propose an advanced method based on symbolic execution and historical exact scripts.

### *A. Naive Solution: Address-based Method*

As mentioned in §II-B, the Bitcoin community defines seven standard types of scripts. The semantics of standard scripts can be learned by regular expressions and decoded Bitcoin addresses. Bitcoin clients provide the interface to calculate the address of the standard scripts. Using the interface, we can check whether a script is with simple-key or impossible-key through addresses by enumerating simple and impossible keys. We can also check whether a script is a non-zero Nulldata script. We name this solution as the address-based method and also implement it in the later evaluation.

This naive solution has two shortages. First, it cannot deal with non-standard scripts. Thus it cannot cover all the defects. Second, as mentioned in §II-B, the Pay-to-Script-Hash (P2SH) and Pay-to-Witness-Script-Hash (P2WSH) scripts only record the hash of the exact executing script. Therefore, for a P2SH or P2WSH script, the address-based method and even the previous symbolic tool [29] cannot work. Hence, we propose a method based on symbolic execution and historical transactions to solve these two problems.

### *B. Overview*

Figure 5 gives an overview of our defect detection approach that consists of the following four steps.

**Conducting Exact Script DB.** Since a Bitcoin transaction (TX) input includes a pointer to a previous TX output, if we want to get the script from which the bitcoins are paid, we need to refer to the previous TX output, especially for the P2SH/P2WSH script that splits into 2 TXs. We design a cache-based method to process the raw TXs to a database of

TABLE III
WORDS IN EXPRESSION OF SYMBOLIC PATTERNS

| Words | Description |
|---|---|
| $S_{all}$ | All generated symbolic stacks. |
| $S_i$ | The $i$ symbolic stack. |
| $B_{all}$ | All branches after symbolic execution. |
| $B_i$ | The $i$ branch of the final branches. |
| $Constraints_b$ | A list of constraints for branch $b$. |
| $SAT$<br>$UNSAT$ | Whether there is a satisfying assignment or not. |
| $Model_b$ | Satisfying assignment of $Constraints_b$. |
| $M_k$ | The $k$ assignment in model $M$. |
| $M_k.Parameters$ | The needed parameters of $M_k$. |
| $M_k.Type$ | The symbol type of $M_k$. |
| $M_k.Value$ | The assigned value of $M_k$. |
| $txid$ | The id of the spending transaction. |
| $SigTypes$ | The symbol types of $CHECKSIG$ and $CHECKMULTISIG$. |

the P2SH/P2WSH outputs. It caches the latest UTXOs in the memory to speed up processing.

**Finding Exact Script.** Once the same P2SH or P2WSH script has been ever spent, we obtain the exact script by looking up and decoding the input script, of which the UTXO script is the same as the evaluating script. In addition, there are several P2SH-P2WSH-* scripts in Bitcoin; thus, we first decode P2SH and then P2WSH. The scripts which are not P2SH or P2WSH are regarded as an exact script for the symbolic engine.

**Symbolic Execution.** The exact script will be executed in our symbolic engine. The symbolic engine uses the symbolic stacks to execute the script. The parameters of the transactions and some non-linear operation results are also provided as symbols. For each conditional operation, it generates a new symbolic state. Finally, it obtains several branches and corresponding constraints as the execution results. After that, the constraints are checked by the SMT Solver to decide whether the branch is possible. The details of the symbolic engine for Bitcoin scripts are described in § IV-C.

**Pattern Matching.** Based on the execution results, including the symbolic branches, constraints, and models, we use symbolic patterns to detect the defects. A script is regarded as having a defect if the symbolic execution result matches the pattern.

Note that in the following we refer to this advanced method simply as the symbolic method. It includes not only symbolic execution but also the historical exact script enhancement and defect patterns.

### C. Symbolic Engine

The main idea of symbolic execution is to use symbols to replace the unknown input of the program, thereby knowing the execution semantics of the program, using SMT solvers or other methods for further judgment. We set the following unknown variables as symbols: stack given by the unknown input script, transaction parameters, results of non-linear operations, etc. We describe our symbolic engine as follows.

**Architecture.** The symbolic engine holds a list of stacks ("state" in traditional symbolic execution). Each stack is equipped with its constraints and current program counter. The symbolic engine executes operations for each stack and then keeps their final status for further detection.

**Operational Semantics.** The operational semantics can be divided into "Flow-control", "Stack", "Splice", "Bitwise-logic", "Arithmetic", "Crypto", "Locktime", "Reserved-words", defined in the community wiki [17], and our implementation follows these semantics.

**Symbolic Bitcoin Stack.** As mentioned in Section II, the output script is executed after the input script. In this case, the input script might leave some unknown stacks for the output script. Therefore, those left stacks are provided as symbols if needed for execution.

**Transaction Parameter.** In the Bitcoin script engine, several operation codes support the script to obtain the parameters of the transaction input: $nLockTime$, $nSequence$, and $txid$. Therefore, these parameters are provided as symbols if needed for execution.

**Non-linear Operation.** There are also several hash functions and signature verification functions in Bitcoin (e.g., $HASH160$, $SHA256$, $CHECKSIG$, etc.). As for these non-linear operations, the SMT solver (Z3) cannot check the non-linear constraints. Hence, the operation results are provided as symbols. And the parameters (e.g., pre-image, public-key, etc.) are attached to these symbols for further pattern matching. In this way, although the symbolic engine cannot exactly act as the actual non-linear operation, the details of the operation are preserved for further detection.

**Conditional Operation.** Several Bitcoin operation codes will trigger conditional operation, e.g., $EUQAL$, $GREATERTHAN$, $IF$, etc. When executing such operation codes, the symbolic engine will come out with two symbolic states with different constraints appended. And then, both branches continue the next operation.

**Spendable Constraint.** Finally, each branch appends a constraint that the top of the stack is true. This is the spendable condition of the Bitcoin script. In this case, the branches can be checked by the SMT solver to find whether they can spend the Bitcoin.

### D. Symbolic Patterns

After the symbolic execution of the specific Bitcoin script, we obtain the results, including symbolic stacks, branches, constraints, and models. Based on these results, we use the following patterns to check the defects. The patterns are presented as several symbolic expressions, in which the terms are described in Table III.

**Unbinded-txid.** As shown in Eqn. (1), if there is a branch $B_i$ that can find the satisfying assignments, and all the assignments ($M_k$) in $Model_{B_i}$ do not need the txid (the parameters of $M_k$ do not include txid), then this script is with the unbinded-txid defect.

$$\begin{aligned} &\exists B_i \in B_{all} \rightarrow Constraints_{B_i} \mapsto SAT, \\ &\forall M_k \in Model_{B_i} \rightarrow txid \notin M_k.Parameters \end{aligned} \tag{1}$$

**Simple-key.** As shown in Eqn. (2), if there is a spendable branch $B_i$ that is using the public-key ($M_k.Pubkey$) that

belongs to a dictionary of the public-key of simple private-keys ($SimpleKeys$), then the script is considered to have the simple-key defect.

$$\begin{aligned} &\exists B_i \in B_{all} \rightarrow Constraints_{B_i} \mapsto SAT, \\ &\exists M_k \in Model_{B_i} \rightarrow M_k.Pubkey \in SimpleKeys \end{aligned} \tag{2}$$

**Useless-sig.** As shown in Eqn. (3), if there is a spendable branch $B_i$ and one of its satisfying assignments ($M_k$) is to assign the symbol of $SigTypes$ to 0, then this script uses useless-sig, leading to defect.

$$\begin{aligned} &\exists B_i \in B_{all} \rightarrow Constraints_{B_i} \mapsto SAT, \\ &\exists M_k \in Model_{B_i} \rightarrow \begin{cases} M_k.Type \in SigTypes, \\ M_k.Value = 0 \end{cases} \end{aligned} \tag{3}$$

**Uncertain-sig.** As shown in Eqn. (4), if there is a spendable branch $B_i$ that one of its satisfying assignments is related to the symbol of $SigType$, and the parameter of this symbol depends on the symbolic stack $S_i$, then this script is with the uncertain-sig defect. The symbolic stack $S_i$ refers to a result stack left by the input script. Hence, for example, once a public-key of "CHECKSIG" in the output script depends on such $S_i$, the parameter (public-key) is given by the input script instead of the output script. Therefore, the output script uses "CHECKSIG" without giving the parameter (public-key) by itself, but by the input script.

$$\begin{aligned} &\exists B_i \in B_{all} \rightarrow Constraints_{B_i} \mapsto SAT, \\ &\exists M_k \in Model_{B_i} \rightarrow \begin{cases} M_k.Type \in SigTypes, \\ \exists S_j \in S_{all}, S_j \in M_k.Parameter \end{cases} \end{aligned} \tag{4}$$

**Impossible-key.** The pattern of impossible-key is shown in Eqn. (5). For a Bitcoin script with an impossible-key defect, it must have a spendable branch $B_i$ that all the public-keys $M_k.Pubkey$ in the parameters of the assigned symbols are impossible. We define the "IsPossible()" function by checking whether more than half of the bytes of the public-key or the public-key-hash are the same, returning false if so. In Bitcoin, the public-key is generated from the private-key by cryptographic algorithms. Therefore, the bytes in the public-key are guaranteed to be random and uniform, so as the bytes in its hash value (public-key-hash). If a user wants a certain form of public-key (e.g., prefixed with "2022..." or "abcd..."), he can only try a lot of private-keys. Since the probability of each byte is random and uniform, for a certain byte (e.g., "0x22"), the more times it appears in the public-key (e.g., "0x22222222..."), the lower the probability of generating this public-key. Hence, once half of the bytes are the same in such a 256-bit public-key ($\frac{256}{2} = 128$ bits), the probability of generating such a key is $\frac{1}{2^{128}}$, which is nearly impossible. In addition, for the remaining branches $B_j$ that do not have the above characteristic, they must be unsatisfying. If the script is matched with this pattern, then it is considered to have an impossible-key defect.

$$\begin{aligned} &\exists B_i \in B_{all} \rightarrow Constraints_{B_i} \mapsto SAT, \\ &\forall M_k \in Model_{B_i} \rightarrow IsPossible(M_k.Pubkey) = False, \\ &\forall B_j \in (B_{all} - \{B_i\}) \rightarrow Constraints_{B_i} \mapsto UNSAT, \end{aligned} \tag{5}$$

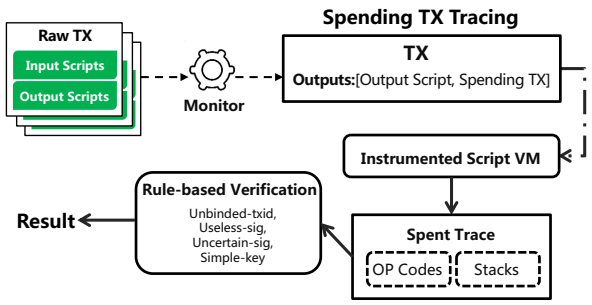



Fig. 6. Overview of Exploitation Tracing

**Never-true.** As shown in Eqn. (6), if all the branches have no satisfying assignment, the script is regarded as a never-true script.

$$\forall B_i \in B_{all} \rightarrow Constraints_{B_i} \mapsto UNSAT \tag{6}$$

## V. Exploitation Tracing

In order to verify and trace the exploitation of the detected defects, we propose the exploitation tracing method.

### A. Overview

Figure 6 shows the overview of the exploitation tracing method. It consists of the following steps.

**Spending TX Tracing.** First, we use an on-chain monitor to trace the inputs and outputs of each raw transaction (TX). Once it finds that a detected buggy script is being spent, it will trace this spending transaction. Hence a trace is represented as a tuple of an output script and a spending transaction.

**Instrumented Script VM.** We instrument the Bitcoin VM to record the traces of the spending transactions. Towards each transaction, it traces the operation (OP) codes and stacks for each step of execution. The trace is like the example shown in Table II.

**Rule-based Verification.** We define several exploiting rules to check whether the exact buggy scripts are exploited. The rules are case by case, corresponding to the four proposed attacker-spendable defects (which will be introduced in the next subsection). Note that there are no exploiting rules for the never-spendable defects. Because the never-spendable outputs theoretically will not be spent. In other words, once there are any spent traces of the never-spendable outputs, then our detection will be proved wrong. In this way, we can find whether the real-world execution paths exploit the defects.

### B. Exploiting Rules

Corresponding to the definition in §III, the exploiting rules are as follows.

**Unbinded-txid.** During the whole execution of the spending buggy output, the VM does not operate $CHECKSIG$, $CHECKMULTISIG$, $CHECKSIGVERIFY$, or $CHECKMULTISIGVERIFY$. If so, the unbinded-txid defect in the previous output is regarded as exploited.

**Simple-key.** Once the VM loads a simple key to the stack for $CHECKSIG*$ operations, the simple-key defect in the spending output is regarded as exploited.

**Useless-sig.** After executing $CHECKSIG$ or $CHECKMULTISIG$ operations, the top of the stacks is $FALSE$. It means that the signature is wrong, but the output is still successfully spent (exploited) on the Bitcoin blockchain.

**Uncertain-sig.** When executing $CHECKSIG*$ operations, the key parameters in the stacks are not from the output scripts, including the value, number, or hash value of the public key and signature. If so, the uncertain-sig defect in the previous output is regarded as exploited.

More details of the rules can be found in our source code at UnsafeBTC.com.

## VI. Implementation and Evaluation

This section first describes our implementation of BSHUNTER. And then, it introduces our empirical study of detecting and tracing defects in real-world Bitcoin scripts in order to help Bitcoin users/developers to prevent losing their bitcoins.

**Implementation.** First, we implement the detecting and tracing methods partially on BTCD v0.20.0-beta [27], one of the Golang clients of the Bitcoin community. The added code in exact script extraction and on-chain tracing is about 2,000 lines. Second, we implement the detecting and tracing methods partially in Python. In the meanwhile, we also implement the address-based method for comparison. As for the symbolic engine of Bitcoin scripts, we use Z3 [20] to generate the symbol and solve the SMT problems. The Python code in defect detection and tracing is about 2,500 lines. In BSHunter, the looking up of exact scripts is conducted with a key-value dictionary in Python. The key is the address of the output script (P2SH/P2WSH), and the value is the first transaction (TX) that spends it. In this way, as for a specific P2SH/P2WSH output script, we can find its first spending TX (the TX hash). And then, in the Bitcoin full node, we query this TX hash to get and decode the TX itself. In this way, we can get the input script. As for the decoding, we decode the last item of the input script as its exact script for P2SH. We also decode the last item of the witnesses for P2WSH. This is just a simplified explanation. Actually, in our implementation, we add caches to make it faster. The detailed decoding codes have been open-source[2]. In addition, the exact scripts on UnsafeBTC.com have already been extracted and decoded in order to ease the validation.

**Experimental Setup.** All the experiments below are conducted on a CentOs server, which is equipped with 1TB Samsung SSD storage and an Intel i7-5820K CPU. Before evaluation, it takes six days for our server to synchronize with the Bitcoin blockchain to the highest block. Finally, we synchronized the data of Bitcoin from 0 to 749,000 blocks (as of August 11, 2022).

[2]https://github.com/InPlusLab/bshunter-btcd

**Research Questions.** Based on BSHUNTER, we conduct extensive experiments to answer three questions in the following subsections:

**RQ1:** What is the effectiveness of our proposed methods for defect detection?

**RQ2:** What is the exploitation status of Bitcoin scripts with defects?

**RQ3:** Where are buggy scripts from, and how to avoid them?

### A. RQ1: Effectiveness of Defect Detection Methods

We evaluate the effectiveness of our defect detection method of BSHunter in two ways: (1) the number of scripts that it can support in RQ1-1; and (2) its advantage over the address-based method in RQ1-2.

**RQ1-1: How many scripts can BSHunter support?**

After the extraction of Bitcoin scripts, we obtain 2,005,704,690 output scripts in total. We run BSHUNTER to execute these scripts to see the feasibility. As a result, the proposed symbolic engine can support all of them except for 498,735 output scripts. The unsupported reasons are as follows. **(1) Path explosion.** 924 unique scripts that use too many "IF" and "CHECKSIG" operations generate a large number of branches during symbolic execution, leading to the path explosion problem that the engine cannot finish the execution. **(2) Uncertain stack movement.** Bitcoin supports moving the "n" item of the stack using $PICK$ and $ROLL$. There are 2 scripts where the "n" is a symbol that the symbolic engine cannot execute it. **(3) Latest Taproot soft-fork.** Bitcoin activated the Taproot soft-fork at the end of 2021 [31]. In our experiment, 497,809 unique scripts use the Taproot, which is not supported by BSHUNTER now. The Taproot output script (P2TR) also records the hash of the exact script but only shows partial branches of the exact script when being spent. Thus one P2TR script might have different branches in different historical transactions. But currently, our tool can only extract the scripts in a single historical transaction for P2SH/P2WSH. Therefore, we need further efforts to support script extraction in multiple transactions for one P2TR.

Note that we do not use the coverage of code for evaluation. The reason is that the Bitcoin script does not support the JUMP or GOTO operation as other script languages. Its symbolic execution must be from the beginning to the end or broken by some unexpected errors. Hence, the coverage for a specific script is either 100% or 0%, which is useless for evaluation.

**RQ1-2: Is the symbolic method better than the address-based method?**

**True Positives.** Table IV shows that 383,544 scripts are detected as positives. We check them in two ways: manual checking and exploitation tracing. For the former, we first use the address-based method to confirm 268,015 scripts. Then, we filter out the scripts that do not include the "CHECKSIG*" operations (obviously unbinded-txid). Note that the filtering is not provided by the address interface of the Bitcoin clients, thus not counted as the address-based method. For the remaining unique scripts, we manually check them. We found all

TABLE IV
COMPARISON OF VULNERABLE SCRIPTS DETECTED BY SYMBOLIC METHOD AND ADDRESS-BASED METHOD (RQ1-2)

| Metric | Defect | Address* | Symbolic* | Gain |
|---|---|---|---|---|
| Count | Unbinded-txid | 0 | 77,141 | 77,141 |
| | Simple-key | 1,484 | 1,484 | 0 |
| | Useless-sig | 0 | 38,284 | 38,284 |
| | Uncertain-sig | 0 | 43 | 43 |
| | Impossible-key | 205,284 | 205,320 | 36 |
| | Never-true | 61,247 | 61,272 | 25 |
| | **Total** | **268,015** | **383,544** | **+43%** |
| BTC | Unbinded-txid | 0 | 55.67 | 55.67 |
| | Simple-key | 21.89 | 21.89 | 0 |
| | Useless-sig | 0 | 0.21 | 0.21 |
| | Uncertain-sig | 0 | 0.25 | 0.25 |
| | Impossible-key | 392.92 | 3,002.28 | 2609.36 |
| | Never-true | 35.115 | 35.120 | 0.005 |
| | **Total** | **449.92** | **3,115.43** | **+592%** |

detected positives are correct, which means there are 0 false positives. For the latter, we trace the spending transactions for the positives. This provides real-world exploitation evidences, which will be described in the next subsection (RQ2). All evidences are available on our website. The threat is discussed in §VII.

**Comparison.** Table IV shows the comparison of scripts detected by the symbolic method and address-based method. In this table, we measure the count and amount of bitcoins (BTC) for different defects. The result of the symbolic method includes not only the buggy scripts matched by the address-based method, but also more non-standard buggy scripts. We use "Gain" to evaluate the increment from the non-standard scripts in this table. We can observe that, for all defects except simple-key, the symbolic method has improved to varying degrees compared with the address-based method. Moreover, the symbolic method can detect the unbinded-txid, useless-sig, and uncertain-sig scripts, while the address-based method cannot. The reason is that these three defects do not exist in standard scripts. The threat to this comparison is also given in §VII.

**Answer to RQ1**. In total, the count of defects detected by the proposed method is increased by 43% compared with the address-based method. The amount of BTC of the defects detected by the proposed method is increased by 592% compared with the address-based method. The largest part of the increment (2609.36 BTC) comes from the 23 impossible-key scripts, which will be studied empirically as well as other typical scripts in the RQ3.

### *B. RQ2: Exploitation Tracing of Buggy Scripts*

Next, we will trace the detected buggy Bitcoin scripts to answer **RQ2**. We study the buggy scripts in view of spent status (RQ2-1) and exploitation evidences (RQ2-2).

**RQ2-1: How many buggy scripts have been spent?**

We use our proposed exploitation tracing method to see their status. The results are shown in Table V.

TABLE V
SPENT STATUS OF DETECTED BUGGY SCRIPTS (RQ2-1)

| Defect | # Spent (BTC) | # Unspent (BTC) |
|---|---|---|
| Unbinded-txid | 76,580 (54.02) | 561 (1.65) |
| Simple-key | 1,484 (21.89) | 0 |
| Useless-sig | 37,782 (0.206) | 502 (0.007) |
| Uncertain-sig | 43 (0.25) | 0 |
| Impossible-key | 0 | 205,320 (3,002.28) |
| Never-true | 0 | 61,272 (35.12) |

TABLE VI
TRACES OF SPENT ATTACKER-SPENDABLE SCRIPTS (RQ2-2)

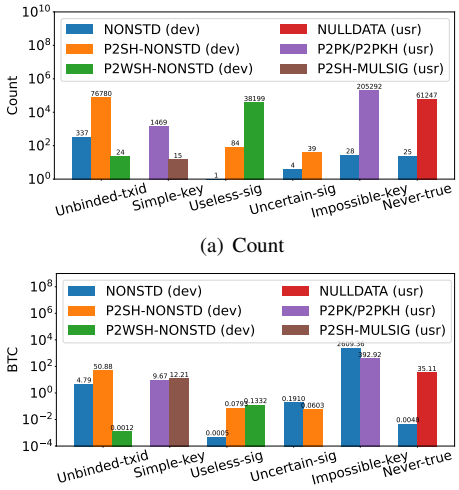
| Defect | # Exploited (BTC) | # Non-Ex' (BTC) |
|---|---|---|
| Unbinded-txid | 76,566 (53.82) | 14 (0.20) |
| Simple-key | 1,484 (21.89) | 0 |
| Useless-sig | 6,045 (0.067) | 31,737 (0.139) |
| Uncertain-sig | 35 (0.249) | 8 (0.001) |

**Attacker-spendable.** As for the attacker-spendable scripts, most of them have been spent. Especially all the simple-key and uncertain-sig scripts have been spent. The amount of the spent scripts is 76.37 BTC in total, which is around $1,500,000 now. We also check the remaining unspent attacker-spendable scripts for why they remain. As for the 1.65 BTC in unbinded-txid scripts, 1 BTC is in a script that requires the input for a pre-image of a hash value, which is unknown to attackers. However, once the owner wants to spend the script, it will broadcast the input to the network then the attackers can steal the 1 BTC, as described in Figure 3. The other 0.65 BTC in unbinded-txid is divided into 560 scripts with small amounts. Some of them are even not enough to pay the transaction fees and thus remain. As for the 502 unspent useless-sig scripts, 3 of them are the same as the examples in §III-C. The other 499 scripts are all like "PUSH <pubkey> CHECKSIG IFDUP NOTIF 16 CHECKSEQUENCEVERIFY ENDIF", and their amounts are all smaller than 0.00001 BTC, which is vulnerable but not so valuable for attacks. These are mainly caused by the lightning network [39] transactions, which could be studied in future work. We will further trace the spending transactions in RQ2-2 to provide more exploitation evidences.

**Never-spendable.** As shown in Table V, none of the impossible-key and never-true scripts as been spent. On the other hand, this also provides evidence that confirms the effectiveness of the proposed methods in detecting never-spendable positives.

**RQ2-2: Are the defects exploited?**

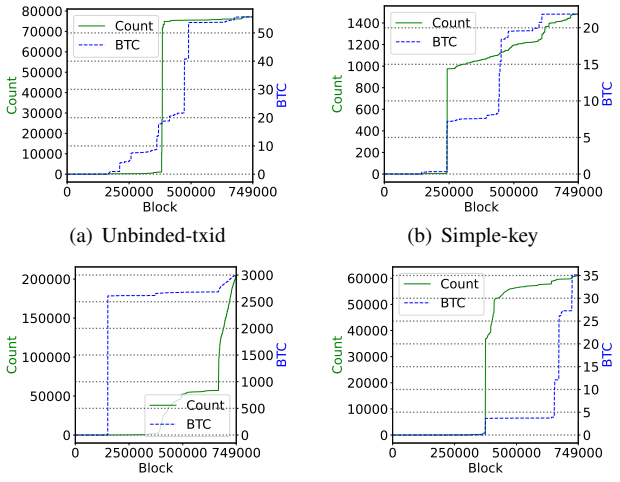
**Exploited Scripts.** As shown in Table VI, almost all the valuable scripts that have defects are exploited. The proportion of exploited amount of bitcoins is $\frac{53.82+21.89+0.067+0.249}{54.02+21.89+0.206+0.25} = \frac{76.026}{76.366} = 99.6\%$. Note that the "Non-Exploited" scripts are not so valuable, indicating why they are not exploited. Since the Bitcoin network is anonymous, we cannot exactly confirm whether they are "stolen" by attackers or owners. However, the exploitation traces of spending paths sincerely provide strong evidences for the detected defects.




(a) Count

(b) BTC

Fig. 7. Distribution of Buggy Scripts ("dev" represents the scripts are mainly by developers, "usr" represents the scripts are mainly by users)




(a) Unbinded-txid

(b) Simple-key

(c) Impossible-key

(d) Never-true

Fig. 8. Typical Buggy Scripts (As of Aug. 2022)

**Answer to RQ2**. As shown in the tracing results, most detected attacker-spendable scripts have been spent. Of the 76.366 BTC worth of spent outputs, a total of up to 99.6% of the BTC amount from the buggy scripts was traced as being exploited. In addition, none of the detected never-spendable scripts has been spent, which laterally proves the effectiveness of the detection method.

### C. RQ3: Empirical Study of Buggy Scripts

We also traverse the historical Bitcoin transactions to answer RQ3.

**Professional Developer v.s. Ordinary User.** First, we provide the distribution of script types in order to learn which type of users the defects mainly come from, as shown in Figure 7. In this figure, the mentioned script types in §II-B are respectively abbreviated as P2PK, P2PKH, MULSIG, P2SH, P2WPKH, P2WSH, NULLDATA, and NONSTD. The Non-standard (NONSTD) scripts can be embedded in the exact scripts of the P2SH and P2WSH scripts. Thus there are P2SH-NONSTD and P2WSH-NONSTD scripts. As the figure shows, the scripts with unbinded-txid, useless-sig, and uncertain-sig are mainly NONSTD-related, in the measurement of both count and amount of bitcoins (BTC). In the meanwhile, for ordinary users, the mainstream Bitcoin wallets (e.g., Bitcoin-core) do not provide a user-familiar feature to conduct the NONSTD-related outputs. Only those developers with relevant Bitcoin knowledge can conduct them. Hence, these scripts are considered from developers. On the contrary, as for the scripts with simple-key, impossible-key, and never-true, it is easy for ordinary users to construct them by wallets. In summary, ordinary users created more buggy scripts than professional developers, but the scripts involved less amount of bitcoins.

**Typical Defects.** Among the six proposed defects, we found that four defects involve the most amount of BTC: unbinded-txid, simple-key, impossible-key, and never-true. Therefore, we provide the cumulative statistics of them to understand the occurrence of buggy scripts in different periods, as shown in Figure 8.

As for the unbinded-txid shown in Figure 8(a), we observe a rapid growth of BTC from block #460000 to #500000. During this period, scripts with a total of 32.06 BTC use "PUSH 1" as the exact executing script, which is spendable by any input script. The address is "3J98t1WpEZ73CNmQviecrnyiWrnqRhWNLy" and unknown users still conduct output scripts to it in February 2023.

As for the simple-key scripts shown in Figure 8(b), the occurrence is continuous. The most used simple private key is the key "1". 1,126 output scripts are controlled by this simple key. As the scripts are standard, they can be represented as the Bitcoin address "1EHNa6Q4Jz2uvNExL497mE43ikXhwF6kZm". This private key's scripts have received a total of 7.82 BTC, which has been fully spent.

As for the impossible-key scripts shown in Figure 8(c), we observe a rapid growth of the amount of bitcoins in the period from block #150000 to #160000. In this period, 23 non-standard scripts are with the code example in §III-E. The amounts of the scripts vary from 21 to 497 [6] BTC, which are counted as 2609.36 BTC in total. The example impossible-key script (as the second output in the link of [4]) that is Non-standard to conduct an address. Hence, they are not detected by the address-based method.

As for the never-true scripts shown in Figure 8(d), we can also observe a rapid growth of BTC from block #650000 to #700000. In this period, 155 valuable nulldata scripts that record a string as "07ffff" joint with an Ethereum address [7]. The total amount of the 155 scripts is 23.329 BTC. Since they return the Ethereum addresses and burn such BTC (over 1 million dollars), we infer that they might be used to mint new tokens on Ethereum (e.g., Non-Fungible Token). However, to the best of our knowledge, there are no related reports.

**Answer to RQ3**. The above observation shows how buggy scripts are generated and exploited. We come out with some suggestions for Bitcoin users and developers: (1) Users should not generate private keys through simple numbers or strings. (2) Users should not carry bitcoins in data storage scripts that are constructed through operations such as $RETURN$. (3) Developers should use verification methods, including symbolic execution, to prevent their scripts from the above defects.

## VII. Threat to Validity

**Validity of Definition.** The impact of Bitcoin script defects depends on our understanding, which could be different for different users/researchers. In this paper, we consider those defects that make the user lose control of payment as our target. However, things go differently once the scenario is not the payment but the storage or others. Moreover, as mentioned, our definitions are inspired by the examples from the community and our own survey, which might not be complete. There might be other uncovered cases where a user loses control. But we claim that the definitions are sound as those defects do make users lose control of bitcoins.

**Validity of Detection.** Our detection method depends on symbolic patterns. Although our experimental results show the feasibility and the true positives, there could be some false negatives. In other words, the detection results are sound but may not be complete. For example, there might be other unknown simple private keys that are not detected. Our method can also be applied to those situations. However, it needs further effort and more computing resources, such as extending the exploitation tracing method to trace all the on-chain transactions.

**Validity of Evaluation.** As shown in Table IV, the major gain of the amount of BTC is impossible-key, while the unbinded-txid and useless-sig only contribute around 56 BTC. As mentioned in §VI-C, the 23 impossible-key scripts caught by BSHUNTER contain a large amount of BTC. Hence it is a threat to the validity of the evaluation.

## VIII. Related Work

**Bitcoin data tools.** BlockSci [28] is a tool for analyzing the raw data from the Bitcoin client. BitcoinETL [33] allows users to export the Bitcoin blockchain data. Google BigQuery [19] is an analytics platform for users to query Bitcoin data. There are many websites providing API for users to get the Bitcoin data, e.g., BTC.COM [14], Blockchain.Info [11], etc. Unfortunately, most tools only provide Bitcoin addresses to users.

**Bitcoin data analysis.** Previous research analyzes Bitcoin in the view of blockchain [26], [24] and transaction [41], [23], [38], [15], [47]. For Bitcoin scripts, Rajput et al. [40] provide a solution to the vulnerability of transaction malleability. Bistarelli et al. [10] study the Nonstandard scripts, and Klomp et al. [29] propose SCRIPTAnalyser as a method of symbolic verification for the Bitcoin scripts. Matzutt et al. [32] analyze the impact of arbitrary content on Bitcoin with more than 1600 files. Strehle et al. [42] also study the Nulldata scripts. Note that previous studies, including the standard examples in Bitcoin Wiki [17], inspire the definition of three defects in this paper. However, none of these studies focus on detecting the defects of Bitcoin scripts. The work most related to us is SCRIPTAnalyser [29], as it also uses symbolic execution. The differences are as follows. First, SCRIPTAnalyser is used for inferring input, not for defect detection or exploitation tracing. Not all the patterns that we define in BSHunter can be detected through the output in SCRIPTAnalyser, e.g., the checking of the private-key or public-key. Hence if we want to apply BSHUNTER on SCRIPTAnalyser, we also need to modify SCRIPTAnalyser for defect detection. However, SCRIPTAnalyser is conducted with Haskell, which is a language so inexperienced to us that we were unable to modify SCRIPTAnalyser. Second, SCRIPTAnalyser does not support those P2SH and P2WSH scripts since it cannot obtain the historical exact scripts. As shown in the monthly statistics on BTC.com, P2SH scripts took up 45% of all scripts in August 2022 [8]. Hence, it cannot detect and trace the defects studied in this paper. Moreover, Bitcoin Wiki [18] also shows the defects of Bitcoin clients instead of Bitcoin scripts. Since those defects cause abnormalities in the client, they have been fixed. However, the defects in this paper DO NOT cause the abnormality in the client. They are executed normally but cause users' loss.

**Smart contract defects.** Although some recent studies examine defects and vulnerabilities in Ethereum smart contracts [25], [30], [37], [22], [16], they cannot be applied to this study because of the differences between Bitcoin scripts and Ethereum contracts. Moreover, the UTXO model in Bitcoin is also very different from the contract model in Ethereum.

## IX. Conclusion

In this paper, we design and develop BSHUNTER to conduct the first systematic study on defects of Bitcoin scripts. First, we define six types of defects in Bitcoin scripts. Second, we propose a symbolic and historical approach to detecting Bitcoin scripts with defects. Third, we design an exploitation tracing method to verify and trace the detected defects. The extensive experimental results demonstrate the feasibility and effectiveness of BSHUNTER. Moreover, we obtained new observations by applying BSHUNTER to all on-chain Bitcoin scripts. We found 383,544 buggy outputs that are related to 3,115.43 BTC in total. We provided empirical tracing results of them.

## Data Availability

We provide the source codes and results on our website https://UnsafeBTC.com.


## Acknowledgement

This work is supported in part by the National Natural Science Foundation of China (62032025), the Technology Program of Guangzhou, China (202103050004), and Hong Kong RGC Projects (PolyU15224121). Zibin Zheng is the corresponding author.

## REFERENCES


[1] *Bitconin Transaction on Blockstream*. http://blockstream.info/tx/a4bfa8ab6435ae5f25dae9d89e4eb67dfa94283ca751f393c1ddc5a837bbc31b.

[2] *Bitconin Transaction on Blockstream*. http://blockstream.info/tx/e6a36e8680933c6ea2226c7fe252e64dced8e624db5f82e5678824085f1e7fdb.

[3] *Bitconin Transaction on Blockstream*. http://blockstream.info/tx/a59012de71dafa1510fd57d339ff488d50da4808c9fd4c001d6de8874d8aa26d.

[4] *Bitconin Transaction on Blockstream*. http://blockstream.info/tx/15ad0894ab42a46eb04108fb8bd66786566a74356d2103f077710733e0516c3a.

[5] *Bitconin Transaction on Blockstream*. http://blockstream.info/tx/ad1209c76f94e2eb80b9929021161adce12498ce919333e50fa1fd4e5f6dead2.

[6] *Bitconin Transaction on Blockstream*. http://blockstream.info/tx/03acfae47d1e0b7674f1193237099d1553d3d8a93ecc85c18c4bec37544fe386.

[7] *Bitconin Transaction on Blockstream*. http://blockstream.info/tx/0122d2f239ae53e8655d83ff0a75bc60f4ffff71554363103fe09787b80cb12b.

[8] *BTC.com Statistics*. https://explorer.btc.com/btc/adapter?type=stats-scripts, 2022.

[9] A. Antonopoulos. *Mastering Bitcoin: Programming the Open Blockchain*. O'Reilly Media, 2017.

[10] Stefano Bistarelli, Ivan Mercanti, and Francesco Santini. An analysis of non-standard bitcoin transactions. In *Crypto Valley Conference on Blockchain Technology*, pages 93–96. IEEE, 2018.

[11] Blockchain.Info. *Website of Blockchain.Info*. https://blockchain.info, 2021.

[12] Joppe W Bos, J Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In *International Conference on Financial Cryptography and Data Security*, pages 157–175. Springer, 2014.

[13] Michael Brengel and Christian Rossow. Identifying key leakage of bitcoin users. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 623–643. Springer, 2018.

[14] BTC.COM. *Website of BTC.COM*. https://btc.com, 2021.

[15] Weili Chen, Jun Wu, Zibin Zheng, Chuan Chen, and Yuren Zhou. Market manipulation of bitcoin: Evidence from mining the mt. gox transaction network. In *2019 IEEE conference on computer communications*, pages 964–972, 2019.

[16] Weili Chen, Zibin Zheng, Edith C-H Ngai, Peilin Zheng, and Yuren Zhou. Exploiting blockchain data to detect smart ponzi schemes on ethereum. *IEEE Access*, 7:37575–37586, 2019.

[17] Bitcoin Community. *Bitcoin Script Examples*. https://en.bitcoin.it/wiki/Script, 2021.

[18] Bitcoin Community. *Common Vulnerabilities and Exposures*. https://en.bitcoin.it/wiki/Common_Vulnerabilities_and_Exposures, 2021.

[19] Allen Day, Evgeny Medvedev, Nirmal AK, and Will Price. *Introducing six new cryptocurrencies in BigQuery Public Datasets—and how to analyze them*. https://cloud.google.com/blog/products/data-analytics/introducing-six-new-cryptocurrencies-in-bigquery-public-datasets-and-how-to-analyze-them, 2018.

[20] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[21] Bitcoin Developer. *Bitcoin transaction stardard*. https://developer.bitcoin.org/devguide/transactions.html, 2020.

[22] João F Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. Smartbugs: a framework to analyze solidity smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1349–1352, 2020.

[23] Michael Fleder, Michael S Kester, and Sudeep Pillai. Bitcoin transaction graph analysis. 2015.

[24] Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert Van Renesse, and Emin Gün Sirer. Decentralization in bitcoin and ethereum networks. 2018.

[25] Alex Groce, Josselin Feist, Gustavo Grieco, and Michael Colburn. What are the actual flaws in important smart contracts (and how can we find them)? In *International Conference on Financial Cryptography and Data Security*, pages 634–653. Springer, 2020.

[26] Jonathan Harvey-Buschel and Can Kisagun. Bitcoin mining decentralization via cost analysis. 2016.

[27] Bitcoin in Go. *BTCD Project*. https://github.com/btcsuite/btcd/, 2021.

[28] Harry Kalodner, Malte Möser, Kevin Lee, Steven Goldfeder, Martin Plattner, Alishah Chator, and Arvind Narayanan. Blocksci: Design and applications of a blockchain analysis platform. In *29th USENIX Security Symposium*, pages 2721–2738, 2020.

[29] Rick Klomp and Andrea Bracciali. On symbolic verification of bitcoin's script language. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 38–56. Springer, 2018.

[30] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269, 2016.

[31] Bitcoin Magazine. *Bitcoin Core 22.0 Released: What's new*. https://bitcoinmagazine.com/technical/new-bitcoin-core-release-taproot, 2021.

[32] Roman Matzutt, Jens Hiller, Martin Henze, Jan Henrik Ziegeldorf, Dirk Müllmann, Oliver Hohlfeld, and Klaus Wehrle. A quantitative analysis of the impact of arbitrary blockchain content on bitcoin. 2018.

[33] Evgeny Medvedev. *Bitcoin ETL*. https://www.github.com/blockchain-etl/bitcoin-etl, 2019.

[34] Walid Mensi, Khamis Hamed Al-Yahyaee, and Sang Hoon Kang. Structural breaks and double long memory of cryptocurrency prices: A comparative analysis from bitcoin and ethereum. volume 29, pages 222–230. Elsevier, 2019.

[35] Malte Möser and Rainer Böhme. Trends, tips, tolls: A longitudinal study of bitcoin transaction fees. In *International Conference on Financial Cryptography and Data Security*, pages 19–33. Springer, 2015.

[36] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*, 2008.

[37] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020.

[38] Micha Ober, Stefan Katzenbeisser, and Kay Hamacher. Structure and anonymity of the bitcoin transaction graph. 2013.

[39] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[40] Ubaidullah Rajput, Fizza Abbas, and Heekuck Oh. A solution towards eliminating transaction malleability in bitcoin. volume 14, 2018.

[41] Dorit Ron and Adi Shamir. Quantitative analysis of the full bitcoin transaction graph. In *International Conference on Financial Cryptography and Data Security*, pages 6–24. Springer, 2013.

[42] Elias Strehle and Fred Steinmetz. Dominating op returns: The impact of omni and veriblock on bitcoin. 2019.

[43] Marie Vasek, Joseph Bonneau, Ryan Castellucci, Cameron Keith, and Tyler Moore. The bitcoin brain drain: Examining the use and abuse of bitcoin brain wallets. In *International Conference on Financial Cryptography and Data Security*, pages 609–618. Springer, 2016.

[44] Marie Vasek and Tyler Moore. Analyzing the bitcoin ponzi scheme ecosystem. In *International Conference on Financial Cryptography and Data Security*, pages 101–112. Springer, 2018.

[45] Ingo Weber, Vincent Gramoli, Alex Ponomarev, Mark Staples, Ralph Holz, An Binh Tran, and Paul Rimba. On availability for blockchain-based systems. In *IEEE 36th Symposium on Reliable Distributed Systems*, 2017.

[46] Bitcoin Wiki. *Proof of burn*, 2021.

[47] Jiajing Wu, Jieli Liu, Weili Chen, Huawei Huang, Zibin Zheng, and Yan Zhang. Detecting mixing services via mining bitcoin transaction network with hybrid motifs. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 52(4):2237–2249, 2021.