Push-Button Verification for BitVM Implementations

HANZHI LIU, Nubit and University of California, Santa Barbara
JINGYU KE, Nubit
HONGBO WEN, Nubit and University of California, Santa Barbara
ROBIN LINUS, ZeroSync and Stanford University
LUKAS GEORGE, ZeroSync
MANISH BISTA, Alpen Labs
HAKAN KARAKUŞ, Chainway Labs
DOMO, Layer 1 Foundation
JUNRUI LIU, University of California, Santa Barbara
YANJU CHEN, University of California, Santa Barbara
YU FENG, Nubit and University of California, Santa Barbara

Bitcoin, while being the most prominent blockchain with the largest market capitalization, suffers from scalability and throughput limitations that impede the development of ecosystem projects like Bitcoin Decentralized Finance (BTCFi). Recent advancements in BitVM propose a promising Layer 2 (L2) solution to enhance Bitcoin's scalability by enabling complex computations off-chain with on-chain verification. However, Bitcoin's constrained programming environment—characterized by its non-Turing-complete Script language lacking loops and recursion, and strict block size limits—makes developing complex applications labor-intensive, error-prone, and necessitates manual partitioning of scripts. Under this complex programming model, subtle mistakes could lead to irreversible damage in a trustless environment like Bitcoin. Ensuring the correctness and security of such programs becomes paramount.

To address these challenges, we introduce the first formal verification tool for BitVM implementations. Our approach involves designing a register-based, higher-level domain-specific language (DSL) that abstracts away complex stack operations, allowing developers to reason about program correctness more effectively while preserving the semantics of the original Bitcoin Script. We present a formal computational model capturing the semantics of BitVM execution and Bitcoin Script, providing a foundation for rigorous verification. To efficiently handle large programs and complex constraints arising from unrolled computations that simulate loops, we summarize repetitive "loop-style" computations using loop invariant predicates in our DSL. We leverage a counterexample-guided inductive synthesis (CEGIS) procedure to lift low-level Bitcoin Script into our DSL, facilitating efficient verification without sacrificing accuracy. Evaluated on 98 benchmarks from BitVM's SNARK verifier, our tool successfully verifies 94% of cases within seconds, demonstrating its effectiveness in enhancing the security and reliability of BitVM.

Additional Key Words and Phrases: BitVM, Bitcoin Script, Formal Verification, Program Synthesis

1 Introduction

Bitcoin, introduced in 2009, is the first and most widely adopted blockchain platform, holding the largest market capitalization among cryptocurrencies. Its robust security model, decentralized governance, and proven resilience have established Bitcoin as a cornerstone in the digital asset ecosystem. Recently, there has been a significant surge in interest to expand Bitcoin's capabilities by building ecosystem projects such as Bitcoin Decentralized Finance (BTCFi), aiming to

Authors' Contact Information: Hanzhi Liu, hanzhi@ucsb.edu, Nubit and University of California, Santa Barbara; Jingyu Ke, windocotber@riema.xyz, Nubit; Hongbo Wen, hongbo@ucsb.edu, Nubit and University of California, Santa Barbara; Robin Linus, roblinus@stanford.edu, ZeroSync and Stanford University; Lukas George, lukas@zerosync.org, ZeroSync; Manish Bista, manish@alpenlabs.io, Alpen Labs; Hakan Karakuş, hakan@chainway.xyz, Chainway Labs; Domo, domodata@proton.me, Layer 1 Foundation; Junrui Liu, junrui@ucsb.edu, University of California, Santa Barbara; Yanju Chen, yanju@ucsb.edu, University of California, Santa Barbara; Yu Feng, yufeng@ucsb.edu, Nubit and University of California, Santa Barbara.

introduce smart contracts and decentralized financial services directly onto the Bitcoin network. However, Bitcoin's inherent scalability and throughput limitations—processing approximately seven transactions per second—pose substantial challenges to such developments.

To address these limitations, and inspired by the success of Layer 2 (L2) solutions [28] in scaling Ethereum, a promising approach is to leverage recent advancements in BitVM [19] to construct an L2 solution for Bitcoin. BitVM is a proposed framework that enables complex computations to be executed off-chain while ensuring their correctness through on-chain verification. By facilitating off-chain computation and minimal on-chain proof verification, BitVM has the potential to significantly enhance Bitcoin's scalability and functionality without requiring changes to the core protocol.

However, developing atop Bitcoin presents unique challenges not encountered in platforms like Ethereum. Ethereum provides a Turing-complete programming model and built-in support for common cryptographic primitives such as elliptic curves and hash functions, enabling developers to write expressive smart contracts efficiently. In contrast, Bitcoin's programming model is highly constrained. The Bitcoin Script language is not Turing-complete and lacks features like loops and recursion, making it cumbersome to express even simple computations. For instance, implementing a standard zero-knowledge SNARK verifier that requires only 200 lines of Solidity code on Ethereum could result in a Bitcoin Script program that is several gigabytes in size. Additionally, spatial constraints arise because large Bitcoin Script programs cannot fit within a single Bitcoin block due to the 4MB block size limit, forcing developers to manually or semi-automatically partition the program into smaller segments. Because programming on BitVM with Bitcoin Script is so complex, it is easy to introduce subtle mistakes that could lead to profound security vulnerabilities. Even small errors in the logic or implementation can result in catastrophic consequences, such as incorrect transaction validation, loss of funds, or the ability for attackers to exploit flaws in the computation, leading to irreversible damage in a trustless environment like Bitcoin. Ensuring the correctness and security of such programs becomes paramount, as these issues can undermine the integrity of the entire system.

Ensuring the correctness of BitVM implementations through formal verification becomes imperative given these challenges. Directly verifying the correctness of low-level Bitcoin Script is very difficult due to two primary reasons. First, the complex low-level stack operations inherent in Bitcoin Script make reasoning about program behavior challenging. Second, the size of the programs, which are typically large due to the unrolling of computations that simulate loops, leads to enormous formulas that are difficult for off-the-shelf constraint solvers [10, 25] to handle efficiently.

Our key insight is based on two observations. First, although the low-level stack operations are hard to reason about, similar to existing work on decompilation, many of them can be lifted to a clean three-address code instruction in a higher-level domain-specific language (DSL). This abstraction simplifies the reasoning process by replacing intricate stack manipulations with more straightforward register-based operations. Second, by studying many complex benchmarks in this domain, we notice that many complex constraints are generated from repetitive computations that simulate the functionality of loops in Bitcoin. Since Bitcoin Script is Turing-incomplete and doesn't support loops, all "loop-style" computations have to be unrolled. If symbolic variables introduced before the loop do not get resolved, they propagate at every iteration, thus bloating the resulting Satisfiability Modulo Theories (SMT) formulas quickly.

Based on the above insight, our solution is motivated by recent successes in program lifting and synthesis [7]. First, we design a register-based, slightly higher-level DSL that abstracts away complex stack operations that are normally orthogonal to the verification tasks. To avoid missing low-level bugs in BitVM, our DSL is carefully designed to maximally preserve the semantics of the original Bitcoin Script. Second, since repetitive "loop-style" computations lead to complex constraints, our DSL provides loop invariant predicates to summarize the effect of the original computations. Third, given the original BitVM implementation in Bitcoin Script as the reference implementation, we leverage a counterexample-guided inductive synthesis (CEGIS) procedure [31] to synthesize an equivalent program in our DSL. In this process, low-level stack operations are lifted to cleaner three-address code versions, and complex "loop-style" operations are summarized and replaced using their loop invariants. The resulting program is then fed to a standard Hoare-style verifier [17], which generates constraints that are much easier for off-the-shelf solvers to handle.

To evaluate our approach, we applied our formal verification tool to the entire BitVM implementation, using a suite of 98 benchmarks derived from the SNARK verifier component of BitVM. Our tool successfully verified 94% of the cases, demonstrating both its effectiveness and practicality. The verification process is efficient, with an average runtime of a few seconds per benchmark. These findings underscore the importance and impact of formal verification in enhancing the security and reliability of blockchain technologies.

In summary, our contributions are as follows:

- **A formal verification tool for BitVM**: We propose the first tool that facilitates formal verification of BitVM implementations, enabling developers to specify and verify correctness properties effectively.
- **A register-based DSL for BitVM**: We design a higher-level DSL that abstracts away complex stack operations and allows for efficient reasoning about BitVM programs while preserving the semantics of the original Bitcoin Script.
- **Efficient handling of repetitive computations**: We introduce a novel approach that utilizes loop invariant predicates to summarize repetitive "loop-style" computations, reducing the complexity of the generated SMT formulas.
- **Program lifting via CEGIS**: We leverage a counterexample-guided inductive synthesis procedure to lift low-level Bitcoin Script to our higher-level DSL, facilitating easier verification without sacrificing correctness.
- **Empirical validation and vulnerability discovery**: Through extensive evaluation, we demonstrate the tool's practicality and its capability to uncover critical vulnerabilities, contributing to the overall security of BitVM.

2 Background

In this section, we provide some background on Bitcoin blockchain and BitVM implementation.

Blockchain and Bitcoin. A blockchain is a decentralized, distributed ledger that records transactions across multiple computers in such a way that the recorded transactions cannot be altered retroactively. This ensures both transparency and security. Bitcoin [21], the first and most well-known cryptocurrency, was introduced in 2009 by an anonymous entity known as Satoshi Nakamoto. It operates on a Proof-of-Work (PoW) consensus mechanism, which ensures the security and integrity of transactions through cryptographic computations performed by network participants (miners).

At a high level, Bitcoin's design focuses on decentralization, immutability, and security. It relies on a chain of blocks, where each block contains a list of transactions and a reference to the previous block, forming a continuous chain. Bitcoin's impact has been profound: as of 2023, Bitcoin handles around 350,000 daily transactions, with a market cap exceeding $500 billion, and an estimated 19 million BTC in circulation. Despite its strong security foundation, the system was primarily designed for simple, trustless value transfers, which limits its capacity for more complex operations and programmability, unlike blockchains like Ethereum [6].

```
pub fn is_positive(depth: u32) -> Script {
    script! {
        { (1 + depth) * Self::N_LIMBS - 1 } OP_PICK
        { Self::HEAD_OFFSET >> 1 }
        OP_LESSTHAN
    }
}
```

Fig. 1. An example code snippet demonstrating a bug in BitVM's implementation.

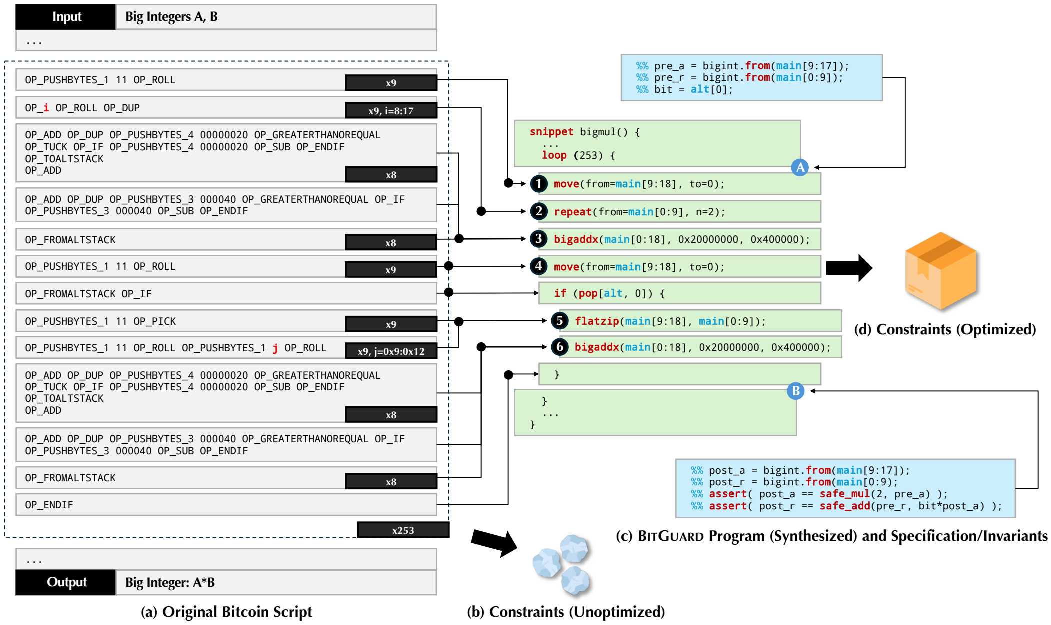
Scaling Bitcoin Through BitVM. Bitcoin's original design comes with significant limitations in terms of throughput and scalability. With an average block size of 1 MB and a block time of roughly 10 minutes, Bitcoin can handle only about 7 transactions per second (TPS)—a far cry from the thousands of TPS supported by traditional payment systems like Visa. This limited throughput, combined with high transaction fees (which can spike during periods of network congestion), makes Bitcoin less suitable for more complex decentralized applications and financial use cases.

To address this, BitVM [19] (invented by Robin Linus) was introduced in 2023 to bring smart contract capabilities to Bitcoin without modifying its consensus rules. Unlike Ethereum, which is Turing-complete and capable of running general-purpose applications directly on-chain, Bitcoin script is intentionally limited for security reasons. BitVM leverages off-chain computation and a prover-verifier model, where complex transactions or computations are done off-chain, and only their validity is checked on-chain. This approach minimizes on-chain data load and ensures scalability while maintaining Bitcoin's security.

However, making BitVM production-ready involves significant complexity. One of the main challenges is compiling high-level domain-specific languages (DSLs) to Bitcoin's low-level, Turing-incomplete script, which demands considerable rewriting of existing cryptographic protocols. For instance, common cryptographic operations, which might be straightforward in languages like Solidity (used on Ethereum), need to be rewritten or optimized to fit Bitcoin's limited script capabilities and resource constraints. Additionally, the space efficiency required by Bitcoin's UTXO (Unspent Transaction Output) model introduces further challenges, demanding innovative techniques to fit within Bitcoin's spatial limitations. Tools like Taproot play a crucial role in enabling BitVM by allowing complex conditions to be verified with minimal on-chain data.

Bugs in BitVM Implementations. The complexity of compiling high-level smart contract logic down to Bitcoin's restrictive script introduces a significant risk of bugs and vulnerabilities, which can lead to severe consequences, particularly in financial systems. Given the manual effort involved in rewriting cryptographic protocols and fitting them into Bitcoin's constraints, there is a high chance of human error during development. These errors can result in vulnerabilities that malicious actors may exploit.

A concrete example of such a bug is the is_positive function used in some BitVM implementations. In one instance, this function was mistakenly designed to treat 0 as a positive number, leading to incorrect behavior in contract execution. The function's logic checks whether a value is positive by examining the most significant limb of the number, but due to a subtle issue in how the offset was calculated, 0 was incorrectly evaluated as positive. Figure 1 shows the faulty code. In this example, a seemingly minor mistake in the arithmetic logic could have led to significant financial losses, depending on how the function was integrated into the broader system. This highlights the critical need for formal verification techniques to ensure the correctness of BitVM implementations.



```
...
OP_PUSHBYTES_1 11 OP_ROLL
OP_i OP_ROLL OP_DUP
OP_ADD OP_DUP OP_PUSHBYTES_4 00000020 OP_GREATERTHANOREQUAL
OP_TUCK OP_IF OP_PUSHBYTES_4 00000020 OP_SUB OP_ENDIF
OP_TOALTSTACK
OP_ADD
OP_ADD OP_DUP OP_PUSHBYTES_3 000040 OP_GREATERTHANOREQUAL OP_IF
OP_PUSHBYTES_3 000040 OP_SUB OP_ENDIF
OP_FROMALTSTACK
OP_PUSHBYTES_1 11 OP_ROLL
OP_FROMALTSTACK OP_IF
OP_PUSHBYTES_1 11 OP_PICK
OP_PUSHBYTES_1 11 OP_ROLL OP_PUSHBYTES_1 j OP_ROLL
OP_ADD OP_DUP OP_PUSHBYTES_4 00000020 OP_GREATERTHANOREQUAL
OP_TUCK OP_IF OP_PUSHBYTES_4 00000020 OP_SUB OP_ENDIF
OP_TOALTSTACK
OP_ADD
OP_ADD OP_DUP OP_PUSHBYTES_3 000040 OP_GREATERTHANOREQUAL OP_IF
OP_PUSHBYTES_3 000040 OP_SUB OP_ENDIF
OP_FROMALTSTACK
OP_ENDIF
...
```

(a) Original Bitcoin Script

(b) Constraints (Unoptimized)

```
%% pre_a = bigint.from(main[9:17]);
%% pre_r = bigint.from(main[0:9]);
%% bit = alt[0];
```

```
snippet bigmul() {
  ...
  loop (253) {
1   move(from=main[9:18], to=0);
2   repeat(from=main[0:9], n=2);
3   bigaddx(main[0:18], 0x20000000, 0x400000);
4   move(from=main[9:18], to=0);
    if (pop[alt, 0]) {
5     flatzip(main[9:18], main[0:9]);
6     bigaddx(main[0:18], 0x20000000, 0x400000);
    }
  }
  ...
}
```

```
%% post_a = bigint.from(main[9:17]);
%% post_r = bigint.from(main[0:9]);
%% assert( post_a == safe_mul(2, pre_a) );
%% assert( post_r == safe_add(pre_r, bit*post_a) );
```

(c) BitGuard Program (Synthesized) and Specification/Invariants

(d) Constraints (Optimized)

Fig. 2. Motivating example.

Without rigorous verification, even small bugs in smart contracts can have disastrous consequences, especially when dealing with high-value financial transactions.

While formal verification is essential for ensuring the correctness of BitVM implementations, it faces several challenges. Bitcoin's scripting language is minimal and lacks the expressiveness of languages like Solidity, making it difficult to model and verify properties. Additionally, BitVM's reliance on off-chain computation and prover-verifier interactions introduces complexity in verifying both on-chain and off-chain components. The manual rewriting of cryptographic protocols to fit Bitcoin's resource constraints further complicates the process. As a result, developing automated formal verification tools that can address these challenges in BitVM requires significant research and specialized techniques.

3 Overview

In this section, we motivate our proposed approach with the aid of a motivating example. To this end, we first present an existing program written in Bitcoin script and then explain the BitGuard workflow.

3.1 Motivating Example

The left-hand-side figure in Figure 2 shows a (partial) Bitcoin script that performs BigInt multiplication in BitVM. BigInt multiplication is a critical cryptographic operation used in blockchain systems, but its implementation in a stack-based virtual machine like BitVM presents several challenges. These difficulties arise from BitVM's low-level stack manipulation and its use of multi-limb arithmetic in loops, which are tricky to understand and verify.

High-Level Overview. The Bitcoin script we analyze performs multi-limb multiplication, where a "limb" represents a chunk or portion of a large integer, treated as a smaller number within a

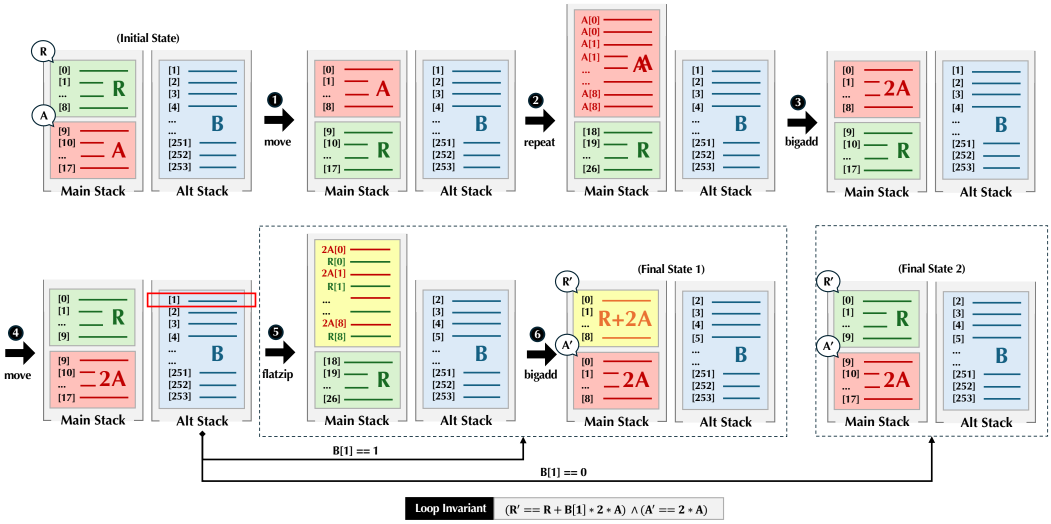


Fig. 3. Illustration of a single iteration step for the computation of big integer multiplication.

larger BigInt. The script iterates through each limb of the multiplicand and multiplier, multiplying them, adding partial products, and storing intermediate results on the stack.

In particular, the multiplication is handled in the following steps:

- **Limb-Based Operations** Each limb of the BigInt is processed separately. A limb is essentially a segment of a large number, typically represented by a fixed number of bits (e.g., 16 or 29 bits). The script multiplies each limb of the first BigInt by corresponding limbs of the second BigInt, with bitwise shifts simulating powers of two. Stack operations like OP_ROLL, OP_DUP, OP_TOALTSTACK/OP_FROMALTSTACK and OP_PICK are used to retrieve, move, and copy limbs between the main stack and alt stack. This makes understanding the operations challenging because stack shuffling makes it hard to track which part of the number is currently being processed.
- **Limb Multiplication and Accumulation** For each bit in a given limb of the multiplier, the script performs a bitwise check (OP_IF) to see if the bit is set to 1. If it is, the corresponding shifted version of the multiplicand is added to the current result. The result and the multiplicand are repeatedly shifted and accumulated on the stack. The intermediate result is updated and stored on the alt stack, while carry values are propagated and handled during the addition of limbs. This process is repeated for all bits in the multiplier's limbs, and the results are combined to form the final product.
- **Stack-Based Computation** The script uses a stack-based architecture, where limbs and intermediate results are stored in and manipulated using the main stack and alt stack. Operations like OP_ROLL are used to cycle elements through the stack, and OP_TOALTSTACK/OP_FROMALTSTACK store and retrieve results. This reliance on stack manipulation makes it difficult to follow the exact flow of control and to verify the correctness of the multiplication algorithm.

Challenges. Although the high-level computation is not complicated, verifying the correctness of this low-level Bitcoin script is particularly challenging for several reasons:

- **Complex Stack Operations** The heavy reliance on low-level stack manipulations such as OP_ROLL and OP_DUP obscures the arithmetic operations being performed. Understanding how the data (limbs) is moved between the main stack and alt stack, and tracking which limb of the BigInt is being operated on, requires careful analysis. Each stack operation shifts the focus to a different part of the BigInt, making it difficult to follow the arithmetic flow.
- **Non-Linear Constraints from Loop Iterations** The script involves multiple loop iterations that process the bits and limbs of the BigInts in sequence. Each iteration involves conditional additions and bitwise shifts, resulting in complex interdependencies between stack operations and arithmetic operations. When unrolled, these loops produce non-linear constraints that are difficult for formal verification tools to resolve efficiently.
- **Carry Propagation** Managing the carry values between limbs adds another layer of complexity. During the multiplication of limbs, intermediate results may produce a carry, which needs to be propagated and added to the next set of operations. Tracking these carry values through the stack-based manipulations makes verification even harder.

Key Insights. To mitigate the complexity of verifying the above Bitcoin script, we leverage two key insights:

- **Lifting Complex Stack Operations to High-Level Register-Based Instructions** One of the main challenges in understanding the script comes from the intricate manipulation of the Bitcoin stack. By lifting these low-level stack operations to higher-level register-based instructions, we can abstract away the complexity of stack shuffling and directly represent operations in a way that is easier to reason about and verify. For instance, operations like OP_ROLL and OP_DUP that manipulate the stack can be lifted to simple register assignments and arithmetic operations, significantly improving clarity.
- **Identifying Loop Patterns and Lifting to Loop Structures** Upon analyzing the script, we observed repetitive patterns where the same chunk of low-level code gets executed multiple times across different iterations. By identifying these loop patterns and lifting them into explicit loop structures, we can avoid unrolling the loops and instead represent them as high-level constructs. This allows us to reason about the loop's behavior more effectively and simplifies the generation of verification conditions. In fact, upon closer inspection, the main body of the original Bitcoin script repeats 253 times, indicating a clear loop structure. By visualizing the core stack operations in Figure 3, we can replace this complex sequence with a corresponding loop invariant that effectively summarizes the key computations.

Our solution: lifting to a high-level DSL. To address these challenges, we propose lifting the original low-level Bitcoin script to a high-level domain-specific language (DSL). Inspired by recent successes in program lifting, our key insight is to synthesize and lift the Bitcoin script into its equivalent high-level representation. The program in the middle of Figure 2 shows the equivalent version in our DSL. Note that this approach abstracts away the complexity of stack manipulation and limb arithmetic by converting the original script into a cleaner, more understandable three-address code format. This snippet also abstracts away the low-level manipulation of individual limbs and carry-bits into a clean loop structure that is easy to verify. Similarly, the main multiplication loop is implemented in a higher-level format.

Applying Hoare logic for verification. Finally, using the synthesized loop invariant, we apply standard Hoare logic to verify the correctness of the program. In particular, given a Hoare triple $\{P\}Q\{R\}$ where P is the precondition, R is the postcondition, Q is the program in our high-level DSL, and loop invariants, we reduce the non-linear constraints generated by the original script into simpler, tractable verification conditions as follows.

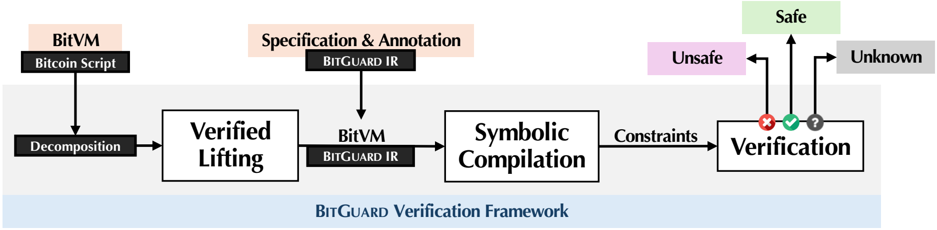


Fig. 4. Framework overview of BitGuard.

- **Precondition** The precondition for the BigInt multiplication in BitVM could involve ensuring that the inputs are valid BigInt values, and that the initial states of the registers (e.g., R and A) are correctly set: $A = \mathsf{BigInt}(A_0) \wedge R = 0$.
- **Postcondition** The postcondition ensures that after the loop has completed, the program has computed the correct product of A and B. I.e., the postcondition describes the final state of R and A after all iterations have completed:

$$R = A \times B \wedge A = 2^k A_0.$$

- **Loop Invariant** After lifting the code to our high-level DSL, we leverage the Houdini algorithm to synthesize the loop invariant—a logical condition that holds true before and after each iteration of a loop. The loop invariant for this multiplication ensures that after each iteration:

$$R' = R + B[1] \times 2A \wedge A' = 2A.$$

  As shown in Figure 3, this loop invariant captures the relationship between the intermediate result R, the bit $B[1]$ from the multiplier, and the multiplicand A.
- **Verification Condition (VC)** The verification conditions are logical formulas that must hold for the program to be considered correct. These conditions are generated from the Hoare triples and are checked to ensure that: a) the precondition implies the invariant holds before the first iteration of the loop:

$$A = \mathsf{BigInt}(A_0) \wedge R = 0 \implies R' = R + B[1] \times 2A \wedge A' = 2A,$$

  b) Invariant holds after each loop iteration, and c) invariant and the loop termination condition imply the postcondition.

These simplified constraints can be verified efficiently using standard formal verification tools, ensuring the correctness of the BigInt multiplication. Note that our approach significantly reduces the complexity of the verification process compared to directly unrolling the original Bitcoin script, making it feasible to formally verify the correctness of complex cryptographic operations like BigInt.

4 The Verification Algorithm

In this section, we introduce the overall verification algorithm built with BitGuard. We first describe a high-level overview of the algorithm, including its key procedures. Then we introduce the domain-specific language, i.e., the BitGuard IR, which can be used to summarize stack-based operations in a verification-friendly way. As an improvement to verification, BitGuard's DSL can be further strengthened by user-provided specification, annotation, and loop invariants, whose integration with BitGuard is then given at the end.

4.1 Algorithm Overview

We show an overview of the algorithm in Figure 4. As illustrated, BitGuard takes as input a Bitcoin script that implements a full system such as BitVM [19], as well as a set of user-provided specification. It then outputs whether the given system is safe regarding the specification, in particular, in three potential outcomes: safe (✔), unsafe (✕) or unknown (?), indicating different level of guarantee that can be provided by BitGuard[1]. Specifically, to reason about the correctness of the given system in an efficient way, BitGuard invokes several critical procedures, as follows:

- **Decomposition** BitGuard first finds a scheme that decomposes the entire system into smaller code snippets. We call such a scheme a *decomposition.* A decomposition can be devised in a semi-automatic manner, which combines both developer's semantic background knowledge and automatic inference from syntactic pattern matching.
- **Verified Lifting** BitGuard then *rewrites* each one or more decomposed code snippets by synthesizing their corresponding snippet in the BitGuard IR. The new snippet is proven semantically equivalent with the original ones, and thus, by assembly of the new snippets, BitGuard gets an equivalent copy of the original system written in the BitGuard IR. We refer to such a procedure as *verified lifting*, and the synthesized program in BitGuard IR is used for further analysis in verification.
- **Verification** With the program in BitGuard IR ready, the user then provides specification and annotations in styles like Hoare logic (i.e., preconditions, postconditions and verification conditions) using BitGuard IR's verification language constructs (e.g., assume and assert). BitGuard then infers invariants for convoluted loops and rewrites them to improve verification efficiency. BitGuard devises a set of symbolic compilation rules that converts a given program into constraints that off-the-shelf solver can consume and reason about.

We elaborate the BitGuard IR in Section 4.2, including its language constructs for modeling typical stack machine behavior such as that of Bitcoin script, and its query and annotation constructs designed for verification. Building on top of the the IR, we describe a procedure for automatic inference of loop invariants given the user-provided specification and annotations in Section 4.3. We then introduce the symbolic compilation and merging rules that convert a BitGuard program into its corresponding constraints in Section 4.4. As the verified lifting procedure is a key procedure, we defer it to Section 5 for more detail.

4.2 The BitGuard Language

A faithful modeling of stack-based languages makes it non-trivial for both machines and humans to reason about. The BitGuard language provides a register-based semantic interface that makes it clear and efficient for program understanding and reasoning. We show the grammar of the BitGuard IR in Figure 5, and elaborate its key designs as follows.

- **High-Level Program and Memory Structure** A BitGuard program is defined by the snippet construct, which itself can be reused in other BitGuard snippets by invocation using expand. As stack-based machines may have more than one stacks for computation (e.g., the main and alt stacks in Bitcoin script), BitGuard models them as special memory locations on the register and provides register-based interface instead.
- **Register-Based Statements and Expressions** Simulated by its register-based memory model, BitGuard allows users to write programs using register-based statements, including basic control flows such as branch and expand, and basic operations with apply, etc. Since loops are either automatically summarized by loop invariants (Section 4.3) or unrolled,

[1] You can find a detailed explanation of their meanings in Section 7.

$$
\begin{array}{llllll}
p & ::= & & & \mathsf{snippet}(i, i*, b) & \textbf{Snippet} \\
s & ::= & & & & \textbf{Statements:} \\
 & \mid & e & & & \text{expression} \\
 & \mid & b & \equiv & \mathsf{block}(s*) & \text{block} \\
 & \mid & g & \equiv & \mathsf{guarded}(e, b) & \text{guarded block} \\
 & \mid & r & \equiv & \mathsf{raw}(z*) & \text{raw block} \\
 & \mid & \sigma & & & \text{stack OP} \\
 & \mid & & & \mathsf{assign}(\varrho, e) & \text{assignment} \\
 & \mid & & & \mathsf{branch}(b, g*) & \text{branch} \\
 & \mid & & & \mathsf{expand}(i, e*) & \text{snippet expansion} \\
 & \mid & & & \mathsf{annotate}(s) & \text{annotation} \\
 & \mid & & & \mathsf{assume}(e) & \text{assumption} \\
 & \mid & & & \mathsf{assert}(e) & \text{assertion} \\
\varrho & ::= & & & & \textbf{Access Paths:} \\
 & \mid & i & & & \text{identifier} \\
 & \mid & o & & + \mid - \mid ... & \text{operators} \\
 & \mid & & & \varrho[e] & \text{list access} \\
 & & & & & \\
z & ::= & \multicolumn{3}{l}{\mathsf{OP\_0} \mid \mathsf{OP\_1} \mid ...} & \textbf{Bitcoin Script OPs}
\end{array}
$$

$$
\begin{array}{llllll}
e & ::= & & & & \textbf{Expressions:} \\
 & \mid & c & & & \text{constant} \\
 & \mid & \varrho & & & \text{access path} \\
 & \mid & k & \equiv & \mu[c] & \text{stack path} \\
 & \mid & \vec{k} & \equiv & \mu[l] & \text{stack paths} \\
 & \mid & l & \equiv & \mathsf{list}(c*) & \text{constant list} \\
 & \mid & & & \mathsf{pop}(\mu) & \text{stack pop} \\
 & \mid & & & \mathsf{apply}(o, e*) & \text{application} \\
\mu & ::= & \multicolumn{3}{l}{\mathsf{main} \mid \mathsf{alt}} & \textbf{Stacks} \\
\sigma & ::= & & & & \textbf{Stack OPs:} \\
 & \mid & & & \mathsf{push}(\mu, e) & \text{stack push} \\
 & \mid & & & \mathsf{move}(\vec{k}, c) & \text{stack move} \\
 & \mid & & & \mathsf{copy}(\vec{k}) & \text{stack copy} \\
 & \mid & & & \mathsf{repeat}(\vec{k}, c) & \text{stack repeat} \\
 & \mid & & & \mathsf{map}(\vec{k}, o, c) & \text{stack map} \\
 & \mid & & & \mathsf{fold}(\vec{k}, o, c) & \text{stack fold} \\
 & \mid & & & \mathsf{zipwith}(\vec{k}, \vec{k}, o) & \text{stack zipwith} \\
 & \mid & & & \mathsf{flatzip}(\vec{k}, \vec{k}) & \text{zip then flatten} \\
 & \mid & & & \mathsf{switch}(\vec{k}) & \text{move bt. stacks}
\end{array}
$$

Fig. 5. The BitGuard language. Note that a symbolic constant is also considered a constant.

BitGuard is free of any explicit loop structures. In addition, access to the memory in BitGuard follows a register-based way, where indices can be provided for reference to a specific location without stack-based restrictions. This effectively removes excessive operations for manipulating stacks during verification, even though BitGuard makes sure each provided register-based language interface has its own equivalent stack-based translation at the backend.

- **Mixed Execution Mode** Recall that BitGuard synthesizes a semantically equivalent snippet for each chunk of decomposed scripts: if synthesis fails, to preserve the full set of semantics of the system, BitGuard transcribes the scripts in their original syntax and preserves them in a raw block, which is then interpreted in a stack compatible mode in BitGuard's symbolic compilation. In addition, BitGuard provides fallback options such as push and pop operators that allow explicit stack management for some special cases when needed.
- **Batched Stack Operations** BitGuard provides a set of major language constructs that summarizes the batched operations performed on stack. For example, moving/duplicating a chunk of stack slots into a different position (move/copy), repeating each element from a chunk of stack slots for multiple times (repeat), zipping two chunks of stack slots by alternating order (flatzip). Such batched stack operations are common patterns frequently observed in real-world projects. Additionally, BitGuard also introduces three standard higher-order stack operators map, fold and zipwith for more customized batch operations.

Query and Verification. To write Hoare-style specification and annotations, BitGuard provides a set of verification operations, namely annotate, assume and assert. Particularly:

- The annotate operator enters a separate *verification scope* where the current program state becomes read-only and more operations from host language/libraries, which may not have stack-based implementation, are allowed for verification purpose. One can write verification conditions and issue queries only such scope.

- The assume operator adds additional conditions to the current path, which correspond to the precondition P in a Hoare triple $\{P\}Q\{R\}$.
- The assert operator issues query for checking in the current path, which correspond to the postcondition R in a Hoare triple $\{P\}Q\{R\}$.

4.3 Inference of Loop Invariants

BitGuard incorporates a Houdini-style inference algorithm that generates conjunctive invariants, which enumerates all possible atomic predicates by unwinding the grammar from Figure 5 up to a fixed bound and then generates the strongest conjunctive invariant over this universe in the standard way [11].

4.4 Symbolic Evaluation for BitGuard Language

In this section, we describe a *symbolic virtual machine* (SVM) [33] with a set of rules that keep track of the program state when evaluating a BitGuard program which, in particular, contains and operates with symbolic values. Figure 6, Figure 7 and Figure 8 show a representative subset of the SVM evaluation rules for the BitGuard language, where a rule written in the following form:

$$\langle x, \gamma, \delta, \pi \rangle \rightsquigarrow \langle y, \gamma', \delta', \pi' \rangle$$

denotes a successful execution of the form x which results in the return form y. In BitGuard's SVM, we use a 4-tuple $\langle p, \gamma, \delta, \pi \rangle$ to describe a program state:

- p is a program counter that points to the immediate next language construct or the evaluated result. We place $\varnothing$ for empty result.
- γ is the *assertion store* that tracks verification conditions generated during the execution. Such conditions can be explicitly produced and pushed to γ via the assert operator, or implicitly added via some operations that implies some facts. For example, access to a list $l[c]$ pushes an implicit condition $c < |l|$ to γ indicating the index c must be smaller than the size of the list l.
- δ is the program store that provides access to the register, memory and stacks during execution. In BitGuard's, a program has access to δ via different access paths (e.g., variable, snippet or stack identifier, index of memory or stack location, etc.) as described by the AccessPaths section in Figure 5.
- π keeps track of the current path condition, which is a boolean value that must evaluate to true in order to reach the current program state. That being said, if a path condition evaluates to false after execution of the form x, then the current program state is *unreachable* and should not be considered anymore; this can also be written as $\langle x, \gamma, \delta, \pi \rangle \rightsquigarrow \bot$.

Symbolic evaluation rules. As shown by Figure 6, BitGuard's SVM starts by identifying the entrance snippet specified by the user with the (Snp) rule, which directs the program counter to each of the statements within the attaching block via the (Blk) rule. The (Exp) rule unrolls the content of a snippet in the current execution context. Rule (Bch) formulates the state transition when branches are met: each if condition is appended to the current path condition when entering its corresponding block, and negated when entering the next branch.

Here, we introduce the notation $\|\pi\|v$ to denote a value v that is obtained under a certain path condition π (aka, a *guarded* value). The (Bch) rule ends with merging of the newly obtained guarded values in program stores that correspond to different branches using the merging operator $\uplus$:

$$\begin{aligned}\delta_0 \uplus \delta_1 = \{&\varrho : \delta_0[\varrho] \mid \varrho \in \Delta(\delta_0, \delta_1)\} \cup \{\varrho : \delta_1[\varrho] \mid \varrho \in \Delta(\delta_1, \delta_0)\} \cup \\ &\{\varrho : \delta_0[\varrho] \cup \delta_1[\varrho] \mid \varrho \in \mathsf{dom}(\delta_0) \land \varrho \in \mathsf{dom}(\delta_1)\}, \\ \text{where } \Delta(\delta_0, \delta_1) = \mathsf{dom}&(\delta_0) \backslash \mathsf{dom}(\delta_1).\end{aligned}$$

$$\frac{\langle b, \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi' \rangle}{\langle \mathsf{snippet}(\_, \_*, b), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi' \rangle}\ (\textsc{Snp})$$

$$\frac{\begin{array}{c}\langle s_0, \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma_0, \delta_0, \pi_0 \rangle \qquad \ldots \\ \langle s_n, \gamma_{n-1}, \delta_{n-1}, \pi_{n-1} \rangle \rightsquigarrow \langle \varnothing, \gamma_n, \delta_n, \pi_n \rangle\end{array}}{\langle \mathsf{block}(s_0, \ldots, s_n), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma_n, \delta_n, \pi_n \rangle}\ (\textsc{Blk})$$

$$\frac{\begin{array}{c}\langle e, \gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \gamma_0, \delta_0, \pi_0 \rangle \qquad \langle \varrho, \gamma_0, \delta_0, \pi_0 \rangle \rightsquigarrow \langle \varrho, \gamma', \delta_0, \pi' \rangle \\ \delta' = \delta_0 \uplus \{\varrho : \{\|\pi'\|v\}\}\end{array}}{\langle \mathsf{assign}(\varrho, e), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi' \rangle}\ (\textsc{Asn})$$

$$\frac{v = \{\|\pi'\|q \in \delta[i] \mid \pi' \rightarrow \pi\}}{\langle i, \gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \gamma, \delta, \pi \rangle}\ (\textsc{Id})$$

$$\frac{\begin{array}{c}\langle \mathsf{main}, \gamma, \delta, \pi \rangle \rightsquigarrow \langle \alpha, \gamma, \delta, \pi \rangle \qquad \langle \mathsf{alt}, \gamma, \delta, \pi \rangle \rightsquigarrow \langle \beta, \gamma, \delta, \pi \rangle \qquad \langle z_0, \gamma, \alpha, \beta, \pi \rangle \hookrightarrow \langle \varnothing, \gamma_0, \alpha_0, \beta_0, \pi_0 \rangle \qquad \ldots \\ \langle z_n, \gamma_{n-1}, \delta_{n-1}, \pi_{n-1} \rangle \hookrightarrow \langle \varnothing, \gamma_n, \alpha_n, \beta_n, \pi_n \rangle \qquad \delta' = \delta \uplus \{\mathsf{main} : \|\pi_n\|\alpha_n\} \uplus \{\mathsf{alt} : \|\pi_n\|\beta_n\}\end{array}}{\langle \mathsf{raw}(z_0, \ldots, z_n), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma_n, \delta', \pi_n \rangle}\ (\textsc{Raw})$$

$$\frac{\begin{array}{c}g_0 \equiv \mathsf{guarded}(e_0, b_0) \qquad \ldots \qquad g_n \equiv \mathsf{guarded}(e_n, b_n) \\ \pi_0 = \pi \qquad \langle e_0, \gamma, \delta, \pi_0 \rangle \rightsquigarrow \langle v_0, \gamma_0, \delta_0, \pi_0 \rangle \qquad \langle b_0, \gamma_0, \delta_0, \pi \wedge v_0 \rangle \rightsquigarrow \langle \varnothing, \gamma_0', \delta_0', \pi_0 \wedge v_0 \rangle \\ \pi_1 = \pi_0 \wedge \neg v_0 \qquad \langle e_1, \gamma, \delta, \pi_1 \rangle \rightsquigarrow \langle v_1, \gamma_1, \delta_1, \pi_1 \rangle \qquad \langle b_1, \gamma_1, \delta_1, \pi_1 \wedge v_1 \rangle \rightsquigarrow \langle \varnothing, \gamma_1', \delta_1', \pi_1 \wedge v_1 \rangle \\ \ldots \\ \pi_n = \pi_{n-1} \wedge \neg v_{n-1} \qquad \langle e_n, \gamma, \delta, \pi_n \rangle \rightsquigarrow \langle v_n, \gamma_n, \delta_n, \pi_n \rangle \qquad \langle b_n, \gamma_n, \delta_n, \pi_n \wedge v_n \rangle \rightsquigarrow \langle \varnothing, \gamma_n', \delta_n', \pi_n \wedge v_n \rangle \\ \pi_b = \pi_n \wedge \neg v_n \qquad \langle b, \gamma, \delta, \pi_b \rangle \rightsquigarrow \langle \varnothing, \gamma_b', \delta_b', \pi_b \rangle \qquad \gamma' = \gamma \cup \gamma_0' \cup \ldots \cup \gamma_n' \cup \gamma_b' \qquad \delta' = \delta \uplus \delta_0' \uplus \ldots \uplus \delta_n' \uplus \delta_b'\end{array}}{\langle \mathsf{branch}(b, g_0, \ldots, g_n), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle}\ (\textsc{Bch})$$

$$\frac{\begin{array}{c}p \equiv \mathsf{snippet}(i, i_0, \ldots, i_n, b) \\ \langle e_0, \gamma, \delta, \pi \rangle \rightsquigarrow \langle v_0, \gamma_0, \delta_0, \pi_0 \rangle \qquad \ldots \qquad \langle e_n, \gamma_{n-1}, \delta_{n-1}, \pi_{n-1} \rangle \rightsquigarrow \langle v_n, \gamma_n, \delta_n, \pi_n \rangle \\ \delta_i = \delta \uplus \{i_0 : \{\|\pi_0\|v_0\}, \ldots, i_n : \{\|\pi_n\|v_n\}\} \qquad \langle p, \gamma_n, \delta_i, \pi_n \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi' \rangle\end{array}}{\langle \mathsf{expand}(i, e_0, \ldots, e_n), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi' \rangle}\ (\textsc{Exp})$$

$$\frac{\begin{array}{c}\langle \varrho, \gamma, \delta, \pi \rangle \rightsquigarrow \langle x, \gamma_0, \delta_0, \pi_0 \rangle \\ \langle e, \gamma_0, \delta_0, \pi_0 \rangle \rightsquigarrow \langle y, \gamma_1, \delta', \pi' \rangle \\ v = \{\|\pi''\|q \in x[y] \mid \pi'' \rightarrow \pi'\} \\ \gamma' = \gamma_1 \cup \{y < |x|\}\end{array}}{\langle \varrho[e], \gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \gamma', \delta', \pi' \rangle}\ (\textsc{Lac})$$

$$\frac{\begin{array}{c}\langle e, \gamma, \delta, \pi \rangle \rightsquigarrow \langle c, \gamma', \delta', \pi' \rangle \\ c \equiv \mathsf{false} \vee \neg\tau_{\mathsf{bool}}(c)\end{array}}{\langle \mathsf{assume}(e), \gamma, \delta, \pi \rangle \rightsquigarrow \bot}\ (\textsc{Asm1})$$

$$\frac{\langle e, \gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \gamma', \delta', \pi' \rangle}{\langle \mathsf{assume}(e), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi' \wedge v \rangle}\ (\textsc{Asm2})$$

$$\frac{\begin{array}{c}\langle e, \gamma, \delta, \pi \rangle \rightsquigarrow \langle c, \gamma', \delta', \pi' \rangle \\ c \equiv \mathsf{false} \vee \neg\tau_{\mathsf{bool}}(c)\end{array}}{\langle \mathsf{assert}(e), \gamma, \delta, \pi \rangle \rightsquigarrow \bot}\ (\textsc{Ast1})$$

$$\frac{\langle e, \gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \gamma', \delta', \pi' \rangle}{\langle \mathsf{assert}(e), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma' \cup \{\pi' \rightarrow v\}, \delta', \pi' \rangle}\ (\textsc{Ast2})$$

$$\frac{\langle s, \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi' \rangle}{\langle \mathsf{annotate}(s), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi' \rangle}\ (\textsc{Ant})$$

Fig. 6. Symbolic evaluation rules (part 1) for statement and access path constructs in BitGuard's symbolic virtual machine.

$$\frac{}{\langle v, \gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \gamma, \delta, \pi \rangle}\ (\textsc{Const})$$

$$\frac{\begin{array}{c}v = [v_0, \ldots, v_n] \qquad v_0 = \{\|\pi\|c_0\} \\ \ldots \qquad v_n = \{\|\pi\|c_n\}\end{array}}{\langle \mathsf{list}(c_0, \ldots, c_n), \gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \gamma, \delta, \pi \rangle}\ (\textsc{List})$$

$$\frac{\mu \in \{\mathsf{main}, \mathsf{alt}\} \qquad l = \delta[\mu]}{\langle \mu, \gamma, \delta, \pi \rangle \rightsquigarrow \langle l, \gamma, \delta, \pi \rangle}\ (\textsc{Sac0})$$

$$\frac{\begin{array}{c}\langle \mu, \gamma, \delta, \pi \rangle \rightsquigarrow \langle l, \gamma, \delta, \pi \rangle \\ v = \{\|\pi'\|q \in l[c] \mid \pi' \rightarrow \pi\} \\ \gamma' = \gamma \cup \{c < |l|\}\end{array}}{\langle \mu[c], \gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \gamma', \delta, \pi \rangle}\ (\textsc{Sac1})$$

$$\frac{\begin{array}{c}l = [c_0, \ldots, c_n] \qquad v = [v_0, \ldots, v_n] \qquad \langle \mu[c_0], \gamma, \delta, \pi \rangle \rightsquigarrow \langle v_0, \gamma_0, \delta, \pi \rangle \\ \ldots \qquad \langle \mu[c_n], \gamma, \delta, \pi \rangle \rightsquigarrow \langle v_n, \gamma_n, \delta, \pi \rangle \qquad \gamma' = \gamma \cup \gamma_0 \cup \ldots \cup \gamma_n\end{array}}{\langle \mu[l], \gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \gamma', \delta, \pi \rangle}\ (\textsc{Sac2})$$

$$\frac{\begin{array}{c}\langle \mu, \gamma, \delta, \pi \rangle \rightsquigarrow \langle l, \gamma, \delta, \pi \rangle \qquad l', v = \overline{\mathsf{pop}}(l) \\ \delta' = \delta \uplus \{\mu : \{\|\pi\|l'\}\} \qquad \gamma' = \gamma \cup \{|l| > 0\}\end{array}}{\langle \mathsf{pop}(\mu), \gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \gamma', \delta', \pi \rangle}\ (\textsc{Pop})$$

$$\frac{\begin{array}{c}o \equiv \oplus \in \{+, -, \ldots\} \qquad \langle e_0, \gamma, \delta, \pi \rangle \rightsquigarrow \langle v_0, \gamma_0, \delta_0, \pi_0 \rangle \\ \ldots \qquad \langle e_n, \gamma_{n-1}, \delta_{n-1}, \pi_{n-1} \rangle \rightsquigarrow \langle v_n, \gamma_n, \delta_n, \pi_n \rangle \qquad v = v_0 \oplus v_1 \oplus \ldots \oplus v_n\end{array}}{\langle \mathsf{apply}(o, e_0, \ldots, e_n), \gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \gamma_n, \delta_n, \pi_n \rangle}\ (\textsc{App})$$

$$\frac{\langle e, \gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \gamma', \delta_0, \pi' \rangle \qquad \langle \mu, \gamma', \delta_0, \pi' \rangle \rightsquigarrow \langle l, \gamma', \delta_0, \pi' \rangle \qquad l' = \overline{\mathsf{push}}(l, v) \qquad \delta' = \delta_0 \uplus \{\mu : \{\|\pi'\|l\}\}}{\langle \mathsf{push}(\mu, e), \gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \gamma', \delta', \pi' \rangle}\ (\textsc{Push})$$

Fig. 7. Symbolic evaluation rules (part 2) for expression and stack operation constructs in BitGuard's symbolic virtual machine. $\overline{\mathsf{pop}}$ is a standard stack operator.

Similar to the (Bch) rule, in the (Asn) rule, any value being assigned to a location also carries the current path condition as its guard and should be merged with the program store δ using the *uplus* operator.

Verification related rules, namely (Asm1), (Asm2), (Ast1) and (Ast2), directly push verification conditions into assertion store γ or path condition π. Such verification conditions need to be implied when a location from the program store δ is accessed. For example, the (Id) rule and (Lac) rule can only access the corresponding values that are guarded by its current path condition.

$$\frac{\begin{array}{c}\langle \mu, \gamma, \delta, \pi \rangle \rightsquigarrow \langle C, \gamma_c, \delta, \pi \rangle \qquad \langle \vec{k}, \gamma, \delta, \pi \rangle \rightsquigarrow \langle A, \gamma_a, \delta, \pi \rangle \\ \vec{k} \equiv \mu[l] \qquad X_0 = \mathsf{delete}(C, l) \qquad X_1 = \mathsf{insert}(X_0, c, A) \\ \delta' = \delta \uplus \{\mu : \{\|\pi\| X_1\}\} \qquad \gamma' = \gamma \cup \gamma_a \cup \gamma_c\end{array}}{\langle \mathsf{move}(\vec{k}, c), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle} \;(\textsc{Move})$$

$$\frac{\begin{array}{c}\langle \mu, \gamma, \delta, \pi \rangle \rightsquigarrow \langle C, \gamma_c, \delta, \pi \rangle \qquad \langle \vec{k}, \gamma, \delta, \pi \rangle \rightsquigarrow \langle A, \gamma_a, \delta, \pi \rangle \\ \vec{k} \equiv \mu[l] \qquad X_0 = \mathsf{insert}(C, 0, A) \\ \delta' = \delta \uplus \{\mu : \{\|\pi\| X_0\}\} \qquad \gamma' = \gamma \cup \gamma_a \cup \gamma_c\end{array}}{\langle \mathsf{copy}(\vec{k}), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle} \;(\textsc{Copy})$$

$$\frac{\begin{array}{c}\langle \mu, \gamma, \delta, \pi \rangle \rightsquigarrow \langle C, \gamma_c, \delta, \pi \rangle \qquad \langle \vec{k}, \gamma, \delta, \pi \rangle \rightsquigarrow \langle A, \gamma_a, \delta, \pi \rangle \\ \vec{k} \equiv \mu[l] \qquad X_0 = \mathsf{repeat}(A, n) \\ X_1 = \mathsf{delete}(C, l) \qquad X_2 = \mathsf{insert}(X_1, 0, X_0) \\ \delta' = \delta \uplus \{\mu : \{\|\pi\| X_2\}\} \qquad \gamma' = \gamma \cup \gamma_a \cup \gamma_k\end{array}}{\langle \mathsf{repeat}(\vec{k}, n), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle} \;(\textsc{Repeat})$$

$$\frac{\begin{array}{c}\langle \mu, \gamma, \delta, \pi \rangle \rightsquigarrow \langle C, \gamma_c, \delta, \pi \rangle \qquad \langle \vec{k}, \gamma, \delta, \pi \rangle \rightsquigarrow \langle A, \gamma_a, \delta, \pi \rangle \\ o \equiv \oplus \in \{+, -, ...\} \qquad \vec{k} \equiv \mu[l] \qquad X_1 = \mathsf{copy}(C) \\ X_0 = [A[0] \oplus c, ..., A[n] \oplus c] \qquad \mathsf{put}(C, l, X_0) \\ \delta' = \delta \uplus \{\mu : \{\|\pi\| X_1\}\} \qquad \gamma' = \gamma \cup \gamma_a \cup \gamma_c\end{array}}{\langle \mathsf{map}(\vec{k}, o, c), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle} \;(\textsc{Map})$$

$$\frac{\begin{array}{c}o \equiv \oplus \in \{+, -, ...\} \qquad \vec{k} \equiv \mu[l] \qquad \langle \mu, \gamma, \delta, \pi \rangle \rightsquigarrow \langle C, \gamma_c, \delta, \pi \rangle \qquad \langle \vec{k}, \gamma, \delta, \pi \rangle \rightsquigarrow \langle A, \gamma_a, \delta, \pi \rangle \qquad x_0 = c \oplus A[0] \qquad x_1 = x_0 \oplus A[1] \\ ... \qquad x_n = x_{n-1} \oplus A[n] \qquad X_0 = \mathsf{delete}(C, l) \qquad X_1 = \mathsf{insert}(X_0, 0, x_n) \qquad \delta' = \delta \uplus \{\mu : \{\|\pi\| X_1\}\} \qquad \gamma' = \gamma \cup \gamma_a \cup \gamma_c\end{array}}{\langle \mathsf{fold}(\vec{k}, o, c), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle} \;(\textsc{Fold})$$

$$\frac{\begin{array}{c}o \equiv \oplus \in \{+, -, ...\} \qquad \vec{k}_a = \mu[l_a] \qquad \vec{k}_b = \mu[l_b] \qquad \langle \mu, \gamma, \delta, \pi \rangle \rightsquigarrow \langle C, \gamma_c, \delta, \pi \rangle \qquad \langle \vec{k}_a, \gamma, \delta, \pi \rangle \rightsquigarrow \langle A, \gamma_a, \delta, \pi \rangle \\ \langle \vec{k}_b, \gamma, \delta, \pi \rangle \rightsquigarrow \langle B, \gamma_b, \delta, \pi \rangle \qquad q \equiv |l_a| == |l_b| \qquad X_0 = [A[0] \oplus B[0], ..., A[n] \oplus B[n]] \\ X_1 = \mathsf{delete}(C, [l_a; l_b]) \qquad X_2 = \mathsf{insert}(X_1, 0, X_0) \qquad \delta' = \delta \uplus \{\mu : \{\|\pi\| X_2\}\} \qquad \gamma' = \gamma \cup \gamma_a \cup \gamma_b \cup \gamma_c \cup q\end{array}}{\langle \mathsf{zipwith}(\vec{k}_a, \vec{k}_b, o), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle} \;(\textsc{ZipWith})$$

$$\frac{\begin{array}{c}\vec{k}_a = \mu[l_a] \qquad \vec{k}_b = \mu[l_b] \qquad \langle \mu, \gamma, \delta, \pi \rangle \rightsquigarrow \langle C, \gamma_c, \delta, \pi \rangle \qquad \langle \vec{k}_a, \gamma, \delta, \pi \rangle \rightsquigarrow \langle A, \gamma_a, \delta, \pi \rangle \qquad \langle \vec{k}_b, \gamma, \delta, \pi \rangle \rightsquigarrow \langle B, \gamma_b, \delta, \pi \rangle \\ q \equiv |l_a| == |l_b| \qquad X_0 = [A[0], B[0], ..., A[n], B[n]] \\ X_1 = \mathsf{delete}(C, [l_a; l_b]) \qquad X_2 = \mathsf{insert}(X_1, 0, X_0) \qquad \delta' = \delta \uplus \{\mu : \{\|\pi\| X_2\}\} \qquad \gamma' = \gamma \cup \gamma_a \cup \gamma_b \cup \gamma_c \cup q\end{array}}{\langle \mathsf{flatzip}(\vec{k}_a, \vec{k}_b), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle} \;(\textsc{FlatZip})$$

$$\frac{\begin{array}{c}\vec{k} \equiv \mu[l] \qquad \mu' = \mathsf{alt\ if}\ \mu \equiv \mathsf{main\ else\ main} \qquad \langle \mu, \gamma, \delta, \pi \rangle \rightsquigarrow \langle C, \gamma_c, \delta, \pi \rangle \qquad \langle \mu', \gamma, \delta, \pi \rangle \rightsquigarrow \langle C', \gamma_c', \delta, \pi \rangle \qquad \langle \vec{k}, \gamma, \delta, \pi \rangle \rightsquigarrow \langle A, \gamma_a, \delta, \pi \rangle \\ X_0 = \mathsf{delete}(C, l) \qquad X_1 = \mathsf{insert}(C', 0, A) \qquad \delta' = \delta \uplus \{\mu : \{\|\pi\| X_0\}\} \uplus \{\mu' : \{\|\pi\| X_1\}\} \qquad \gamma' = \gamma \cup \gamma_a \cup \gamma_c \cup \gamma_c'\end{array}}{\langle \mathsf{switch}(\vec{k}), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle} \;(\textsc{Switch})$$

Fig. 8. Symbolic evaluation rules (part 3) for batched stack operators in BitGuard's symbolic virtual machine. delete, insert, repeat, copy and put are standard operators for list manipulation, whose semantics can be found in libraries such as NumPy [16]; push is a standard stack operator.

We show how BitGuard's expressions and batched stack operators symbolically evaluate and produce results in Figure 7 and Figure 8 respectively. In particular, the (Pop) and (Push) rules allow the program to directly access and operate the stacks, while all the batched stack operators wrap multiple standard stack operations into one. Finally, BitGuard's SVM allows the execution to simulate a stack-based machine by the raw block and its corresponding (Raw) rule: it connects BitGuard's SVM program state with Bitcoin's stack machine which operates with the following form:

$$\langle z, \gamma, \alpha, \beta, \pi \rangle \hookleftarrow \langle \varnothing, \gamma', \alpha', \beta', \pi' \rangle,$$

where α and β corresponds to the main and alt stacks, while z denotes the current opcode. We defer a detailed discussion of it to Appendix A.

5 Verified Lifting via Program Synthesis

Toward lifting low-level Bitcoin Script to a program in BitGuard IR, we implemented a tool for automatically synthesizing BitGuard IR programs from Bitcoin Script implementations.

5.1 Synthesis of BitGuard Snippets

Our tool is based on the counterexample-guided inductive synthesis (CEGIS) paradigm, which leverages both a candidate program generator and a bounded equivalence checker. Specifically, the generator enumerates candidate BitGuard IR programs that aim to match the behavior of the original Bitcoin Script, and the equivalence checker verifies whether the candidate BitGuard IR program is semantically equivalent to the original. If the equivalence check fails, the tool refines the candidate based on counterexamples and repeats the process. This iterative synthesis continues until we either find an equivalent DSL program or exhaust the search space. In the remainder of this section, we provide a detailed overview of the program generator and equivalence checker that form the backbone of our synthesis engine.

5.2 Equivalence Checker

Once a candidate program has been synthesized in our BitGuard IR, it is essential to ensure that the lifted program is semantically equivalent to the original Bitcoin Script program L. However, verifying this equivalence is non-trivial, as there is no off-the-shelf equivalence checker for comparing Bitcoin Script with custom DSLs. To address this challenge, we implemented an equivalence checker to automatically verify the equivalence between a pair of programs: the original Bitcoin Script program L and the corresponding synthesized BitGuard IR program S.

The goal of the equivalence checker is to confirm that the BitGuard IR program S behaves identically to the Bitcoin Script program L, at least within a bounded scope. The core idea is to symbolically evaluate both programs on a common input state and check if their resulting output states are the same. This process is repeated for all relevant input scenarios within a predefined verification bound K, which controls how deeply the equivalence checker explores the behavior of the programs, including how many times loop-style computations are unrolled.

We have implemented our equivalence checker on top of the Rosette framework [33] and leverage its SMT encoding facilities as well as its symbolic evaluation engine. In our implementation, we encode the operational semantics of both the BitGuard IR and Bitcoin Scripts directly within Rosette. Specifically: For DSL programs, we implement the symbolic evaluation rules (Section 4.4) in Rosette. We also formalize the Bitcoin Script Semantics[2], which captures the stack-based nature of Bitcoin Script, including its push and pop operations, arithmetic and logical operators, and control flow constructs.

6 Implementation

We have implemented BitGuard in Racket/Rosette with a back-end constraint solver (Bitwuzla [23] version 0.4.0). The total codebase comprises 2,574 lines of Racket code. The source code of BitGuard is available as open-source software on GitHub.[3] This includes all implementation components and benchmarks of verified Bitcoin scripts. Below, we elaborate on various aspects of our implementation.

Modeling big integers with symbolic limbs. Bitcoin Script represents integers using sign-magnitude representation, where the highest bit serves as the sign bit. During arithmetic operations, numbers are converted to two's complement representation and then converted back after the operation.

To accurately model operations involving big integers (i.e., BigInts) in BitGuard, we introduced a new symbolic operator called `PUSH_BIGINT_X`. This operator allows us to push a large integer onto the symbolic stack, defined by the following parameters:

- **nbits**: The total number of bits of the BigInt.
- **limb_size**: The number of bits per segment (limb).
- **limbs_name**: The base name for each limb.
- **var_name**: The identifier for the entire BigInt.

[2] Since this is not a major contribution of this paper, we defer the discussion of the symbolic evaluation of the Bitcoin script language to Appendix A.

[3] https://github.com/RiemaLabs/pomelo

For example, PUSH_BIGINT_0 254 29 limbs v_0 creates a 254-bit BitInt, split into limbs of 29 bits each, named limbs_0, limbs_1, etc., with a symbolic identifier v_0 for the whole BigInt. The variable v_0 is constrained to be equal to the sum of its limbs, each shifted by its position:

$$v_0 = \sum_{i=0}^{n} \text{limbs}_i \cdot 2^{i \cdot \text{limbs}_{\text{size}}}, \text{ where } n = \left\lceil \frac{\text{nbits}}{\text{limb\_size}} \right\rceil - 1.$$

After this operation, the stack will have limbs_0, limbs_1, ..., limbs_n pushed onto it, where each limbs_i is a symbolic bitvector of size limb_size (except possibly the highest limb, which may be smaller if nbits is not a multiple of limb_size.

Handling sign bits. In our modeling, we handle the sign bit and limb representations carefully. Since in Bitcoin's implementation, each limb of a BigInt is represented as a positive number (with the sign bit being 0 under normal circumstances), we model each limb as a bitvector of size limb_size, and constrain it to be within the range $[0, 2^{\text{limb\_size}} - 1]$.

For the highest limb, we adjust the limb size to account for any remainder bits:

$$\text{head\_limb\_size} = \text{nbits} \mod \text{limb\_size}.$$

If head_limb_size is zero, it means the highest limb is of size limb_size. The sign of a BigInt is determined by checking whether the HEAD_OFFSET-th bit of the highest limb is 1 or 0, where:

$$\text{HEAD\_OFFSET} = \text{nbits} \mod \text{limb\_size}.$$

To ensure that the sign bit is correctly modeled, we constrain the highest limb such that its sign bit is 0 by default. This means we set the most significant bit of the highest limb to 0 in our constraints:

$$\text{limbs\_n}[\text{head\_limb\_size} - 1] = 0.$$

By modeling BigInts in this way, we avoid issues related to sign bits during arithmetic operations. Each limb is treated as an unsigned bitvector, and the entire BigInt is assembled from these limbs.

Abstraction of cryptographic primitives. Cryptographic operations introduce complex non-linear constraints that are difficult for SMT solvers to handle efficiently. We abstracted these primitives using uninterpreted functions with essential properties captured as axioms. For example, hash functions (e.g., OP_SHA256) are modeled as injective functions without specifying their internal workings. This allows the solver to reason about the high-level behavior without dealing with underlying complexities.

7 Evaluation

In this section, we describe the setup and results for our evaluation, which are designed to answer the following key research questions:

- **RQ1 (Performance)** How does BitGuard perform in verification for Bitcoin scripts?
- **RQ2 (Ablation)** How does the key design of BitGuard affect its performance?

Benchmarks. We collect a total of 98 verification tasks from the two major open-source repositories written using Bitcoin script, which contains usage of a wide coverage of Bitcoin script language constructs in various computational tasks, libraries and components, as follows:

- **bitcoin-scriptexec**[4](or **BSE** for short) implements BitVM2 [19], a framework for turing-complete program execution on Bitcoin. It also comes with a library of functions written in Bitcoin script for various computations and operations in arithmetics, cryptography, stack, bitvector, etc.

[4] https://github.com/BitVM/rust-bitcoin-scriptexec

		LOC		
	# of Benchmarks	Avg.	Max	Min
BSE	85	9895	784410	3
BSV	13	86	207	3
Overall	98	8594	784410	3

Table 1. Statistics of the benchmark suite collected.

- **Bitcoin circle STARK verifier**[5](or **BSV** for short) implements a circle plonk [12] verifier in Bitcoin script. It also comes with reusable cryptographic components written in Bitcoin script.

Table 1 shows the key statistics of the collected benchmarks. We formulated 85 benchmarks from BSE and 13 from BSV by extracting corresponding library/component snippets, with specification manually written in BitGuard's query language as Hoare style preconditions and postconditions. Each benchmark has on average 8594 lines of code, with a maximum of 784410 and a minimum of 3 overall. The computation implemented in the benchmarks mainly falls into several categories:

- Big integer operations, including standard bitwise conversion, comparison, arithmetics, etc.
- Elliptic curve (BN254) operations, including standard arithmetics over the curves.
- Merkle tree implementation, including folding and hashing operations used as its building blocks.

Experimental setup. We implemented BitGuard with Rosette [33] on Racket 8.14. All experiments are conducted on an Amazon EC2® instance with an AMD EPYC 7000® CPU, 8 Cores, and 64G of memory running on Ubuntu 20.04. BitGuard encodes semantics of bitcoin script in bitvector theory [4] and sets Bitwuzla [23] as its default backend constraint solver. The default timeout for evaluation of each benchmark is set to 10 minutes.

Evaluation metrics. We use two key metrics to evaluate the performance of BitGuard:

- **Number of Benchmarks Solved** There are three potential outcomes that BitGuard can produce for verification of a benchmark:
  - Safe (denoted by "✔"), meaning that BitGuard cannot find a counterexample that violates the specification;
  - Unsafe (denoted by "✕"), meaning that the BitGuard finds a concrete counterexample that violates the specification;
  - Unknown (denoted by "**?**"), meaning that BitGuard cannot terminate within given time limit, due to various reasons such as complex benchmarks, running out of resource allocation, backend solver giving up, etc.

  To evaluate the effectiveness of our approach, we measure the number of benchmarks with a *known* result (both safe and unsafe are counted) produced by BitGuard as *solved*, as this gives a concrete proof or counterexample as an answer to the given query in the specification. Note that for the benchmarks where loop invariants are inferred, we implement an extra procedure that validates the counterexample proposed by the tool, due to the fact that a loop invariant is an over-approximation of the original loop which introduces false positives when generating counterexamples. BitGuard blocks those counterexamples that are proven false positives and continues with the verification process until a definite conclusion is reached.

[5] https://github.com/Bitcoin-Wildlife-Sanctuary/bitcoin-circle-stark

	Total	Avg. Time	Solved	Safe (✔)	Unsafe (✕)	Unknown (?)
BSE	85	1.37s	79 (93%)	79 (93%)	0 (0%)	6 (7%)
BSV	13	1.36s	13 (100%)	13 (100%)	0 (0%)	0 (0%)
Overall	98	1.36s	92 (94%)	92 (94%)	0 (0%)	6 (6%)

Table 2. Summarized experimental result for performance evaluation of BitGuard.

- **Solving Time** To evaluate the efficiency of our approach, we measure the solving time of benchmarks. In particular, to reduce variance, only the time spent for benchmarks solved are taken into consideration.

7.1 Performance of BitGuard in Verification for Bitcoin scripts (RQ1)

We start by showing the summarized experimental result in Table 2. Overall, out of 98 benchmarks, BitGuard solves 92 (94%) of them, with 92 (94%) of them proven safe (✔) and 0 (0%) of them having counterexamples found, i.e., proven unsafe (✕). BitGuard takes an average of 1.36s to solve a benchmark. Only 6 (6%) of the benchmarks cannot be answered by BitGuard; our analysis shows that the top reasons for producing unknown (?) results are: 1) complex constraints (e.g., mul in bigint), and 2) excessive resource consumption (e.g., sub in bn254/fp254impl).

We show more detail about the status of each benchmark and category in Table 3. For two of the more complex categories, bigint/bits and bigint/inv, BitGuard demonstrates its efficiency. In the bigint/bits category, BitGuard successfully solved 100% of the benchmarks with an average time of 2.81s. Even for programs in the bigint/inv category, which have an average of 131,579 LOCs, BitGuard still managed to solve 33% of the benchmarks, with an average time of 3.89s. There are also some cases that are worth noting, for example, bigint/mul, which contains the most loops, but BitGuard solves it within 144.74s, despite its complexity and the introduction of computationally expensive operations that generate non-linear constraints. However, even though inv_stage1 contains only 1 loop, BitGuard fails to solve it due to the complicated loop invariants.

Failure analysis. For the 6 benchmarks that BitGuard fails to solve, we perform a manual analysis to identify the root causes. A vast majority of them—5 out of 6—could not be solved within the given time limit due to the complex constraints generated by multiple factors, such as the introduction of non-linear operations, complex loop unrolling, and loop invariants. The backend solver gives up on all 6 of them based on its internal strategy. Even after relaxing the time limit to 24 hours, none of these benchmarks could be solved, as they continued to face the same issues related to complex constraints.

Result for RQ1: BitGuard is able to solve a significant portion (92 out of 98, i.e., 94%) of benchmarks with a 1.36s averaged solving time. Therefore, BitGuard is both effective and efficient, and we believe that this answers RQ1 in a positive way.

7.2 Ablation Study (RQ2)

Since there's no publicly available tool for verification of Bitcoin scripts, to evaluate the effectiveness of BitGuard's key design, we conduct an ablation study that compares BitGuard with its baseline version, where a bitcoin script is compiled directly into constraints according to the rules in ??. That is, the baseline version doesn't perform any verified lifting nor optimization. While it still shares the backend solver (Bitwuzla) with the default BitGuard, we refer to this version as Baseline (Bitwuzla).

File	Benchmark	LOC	Result	Time (s)
bigint/ std (BSE)	zip	29	✔	1.48
	copy_zip	29	✔	1.52
	copy_deep	189	✔	1.73
	dup_zip	28	✔	1.53
	copy	38	✔	1.53
	roll	29	✔	1.54
	drop	16	✔	1.54
	is_zero_ke	21	✔	1.50
	is_zero	12	✔	1.54
	is_one_ke	20	✔	1.53
	is_one	11	✔	1.50
	toaltstack	19	✔	1.51
	fromaltstack	28	✔	1.54
	is_negative	3	✔	1.53
	is_positive	13	✔	1.54
	resize	25	✔	1.55
	overall	31	100%	1.54
bigint/ cmp (BSE)	equalverify	21	✔	1.39
	equal	38	✔	1.43
	notequal	39	✔	1.42
	lessthan	31	✔	1.42
	lessthanorequal	32	✔	1.43
	greaterthan	33	✔	1.43
	greaterthanorequal	32	✔	1.43
	overall	32	100%	1.42
bigint/ add (BSE)	double	53	✔	1.66
	add	54	✔	1.66
	add1	44	✔	1.65
	double_allow_overflow	53	✔	1.61
	double_allow_overflow_ke	62	✔	1.64
	double_prevent_overflow	53	✔	1.69
	lshift_prevent_overflow	841	✔	2.85
	add_ref_with_top	65	✔	1.75
	add_ref	56	✔	1.70
	add_ref_stack	85	✔	1.79
	overall	137	100%	1.80
bigint/ bits (BSE)	convert_to_be_bits	1,261	✔	3.40
	convert_to_le_bits	1,015	✔	3.41
	convert_to_be_bits_toaltstack	1,044	✔	3.41
	convert_to_le_bits_toaltstack	1,253	✔	3.4
	limb_from_bytes	70	✔	1.55
	from_bytes	648	✔	1.74
	overall	840	100%	2.81
bigint/ inv (BSE)	div2	474	✔	3.87
	div2rem	474	✔	3.91
	div3	483	?	TO
	div3rem	483	?	TO
	inv_stage1	784,410	?	TO
	inv_stage2	3,150	?	TO
	overall	131,579	33%	3.89
(BSE)	bigint/mul	32,001	✔	144.74
(BSE)	bigint/sub	79	✔	1.56
(BSV)	merkle_tree/verify	82	✔	1.51
bn254/ fp254impl (BSE)	copy	38	✔	1.39
	roll	29	✔	1.39
	drop	16	✔	1.38
	zip	29	✔	1.37
	equal	38	✔	1.42
	equalverify	21	✔	1.40
	convert_to_be_bits	1,270	✔	4.69
	convert_to_be_bits_toaltstack	1,035	✔	3.16
	convert_to_le_bits	1,024	✔	4.41
	convert_to_le_bits_toaltstack	1,253	✔	3.18
	push_modulus	10	✔	1.38
	push_zero	18	✔	1.41
	push_one	7	✔	1.38
	push_one_not_montgomery	10	✔	1.38
	add	65	✔	1.69
	neg	19	✔	1.65
	sub	65	✔	11.75
	double	55	✔	1.56
	is_zero	12	✔	1.39
	is_zero_ke	21	✔	1.40
	is_one_ke	29	✔	1.40
	is_one_ke_not_montgomery	61	✔	1.41
	is_one_not_montgomery	43	✔	1.39
	is_one	11	✔	1.39
	is_field	58	✔	1.38
	square	7,073	?	TO
	div3	483	?	TO
	toaltstack	19	✔	1.37
	fromaltstack	28	✔	1.39
	overall	113	94%	1.75
bn254/ curves (BSE)	push_generator	30	✔	1.4
	push_zero	18	✔	1.39
	is_zero_ke	41	✔	1.41
	neg	41	✔	1.68
	copy	48	✔	1.67
	roll	45	✔	1.59
	drop	30	✔	1.53
	toaltstack	57	✔	1.41
	fromaltstack	84	✔	1.39
	overall	44	100%	5.26
folding (BSV)	check_0_or_1	3	✔	1.36
	decompose_positions_g	207	✔	2.15
	skip_one_and_ext_bits_g	154	✔	1.76
	overall	121	100%	1.76
utils (BSV)	limb_to_le_bits	88	✔	4.13
	ltbbt_exc_low2b	88	✔	3.41
	ltbbt_common	89	✔	4.15
	qm31_reverse	9	✔	1.38
	hint	35	✔	1.39
	ltbbt_exc_low1b	89	✔	3.63
	dup_mv_g	128	✔	1.41
	mv_from_bottom_g	116	✔	1.41
	cta_top_item_first_in_g	30	✔	1.39
	overall	75	100%	2.48

Table 3. Detailed performance of BitGuard on all benchmarks. "_ke" is a shorthand for "_keep_element"; "ltbbt" for "limb_to_be_bits_tas"; "mv" for "m31_vec"; "g" for "gadget".

A subset of benchmarks (21.43%, especially in the category of bn254) is intended for elliptic curve computations over finite fields. Solving such benchmarks generally poses challenges for backend solvers that rely on integer/bitvector theories, as shown in previous works [27]. To explore whether a finite field solver could improve performance, we introduce a second ablative version, Baseline (cvc5/−ff). This version uses cvc5 [2] with specialized finite field theory [25] (i.e., cvc5−ff) as its backend solver. Specifically, for the 21 benchmarks that assume finite field inputs/outputs, BitGuard compiles them into finite field constraints and invokes cvc5−ff; for other benchmarks, cvc5 with default bitvector theory is used.

Figure 9 shows the result for ablation study, where the x-axis represents the total number of benchmarks solved, and the y-axis shows the cumulative time spent. All three configurations show

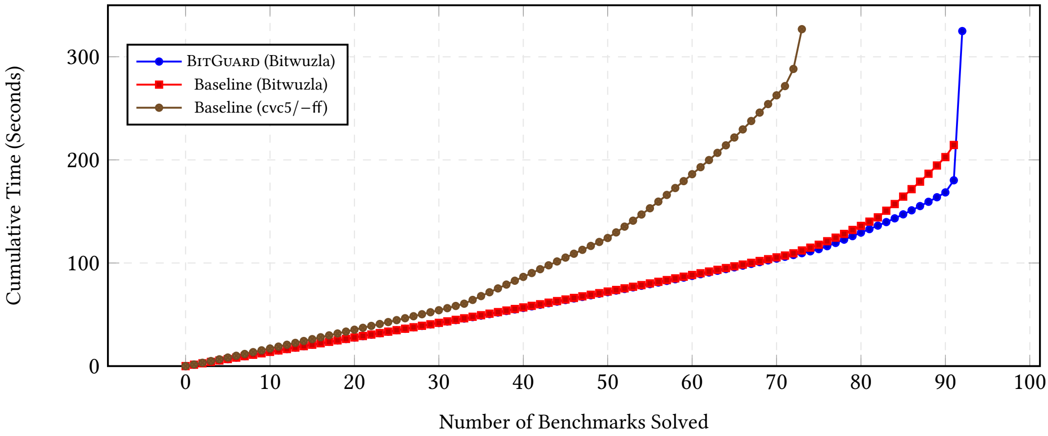


Fig. 9. A comparison between BitGuard and its ablative versions regarding the cumulative time spent as their number of solved benchmarks grows.

an increase in cumulative time as more benchmarks are solved. However, cvc5−ff underperforms compared to both BitGuard and Baseline (Bitwuzla). This is because most benchmarks do not involve direct finite field operations but rather use Bitcoin scripts to simulate these operations. As a result, the finite field optimizations in cvc5−ff do not provide a significant advantage and may even introduce overhead, making it less efficient than the Bitwuzla baseline for this particular set of benchmarks. Compared to Baseline (Bitwuzla), BitGuard demonstrates a clear advantage in 14.29% of benchmarks, thanks to the optimizations provided by the BitGuard DSL. Baseline (cvc5/−ff) initially performs similarly to BitGuard for the first 26 benchmarks but falls behind as more benchmarks are added, with BitGuard ultimately solving 19.39% more benchmarks.

Result for RQ2: BitGuard performs significantly better than its ablative versions, with notable efficiency gains in 14.29% - 67.35% of cases. Thus, BitGuard's design is important for its overall performance, and we believe that this answers RQ2 in a positive way.

8 Related work

Formal methods for cryptography. There is extensive research on applying formal verification techniques to cryptographic protocols. For example, Corin et al.[8] utilized a variant of probabilistic Hoare logic to verify the security of ElGamal, while Gagne et al.[13] applied similar methods to analyze the security of CBC-based MACs, PMAC, and HMAC. Tiwari et al. [32] employed component-based program synthesis to automatically generate padding-based encryption schemes and block cipher modes of operation. EasyCrypt [5] offers a toolset for specifying and proving the correctness of cryptographic protocols.

In addition to the rich literature on the intersection of cryptography and formal methods, there is emerging research on the formal verification of zero-knowledge proofs (ZKPs). Almeida et al.[1] developed a certifying compiler for Σ-protocols, which includes zk-SNARKs, using Isabelle/HOL [24] for formal correctness proofs. Sidorenco et al.[30] produced the first machine-checked proofs for ZK protocols using the Multi-Party Computation-In-The-Head paradigm with EasyCrypt. More recent work has focused on building specialized solvers for polynomial equations over finite fields.

While finite field arithmetic can theoretically be encoded using integer or bitvector theories, solving the resulting constraints with off-the-shelf solvers is often impractical. To address this, Hader et al. [15] developed a custom decision procedure for solving polynomial equations over finite fields by combining a quantifier elimination procedure with Groebner basis computation. Ozdemir et al. [26] recently proposed a finite field solver that does not scale well in our benchmarks due to too many complex constraints. Finally, Coda [20] proposed the first verifier for the functional correctness of ZKP circuits. However, compared to BitGuard, it requires a significant amount of manual effort to write interactive theorem proofs in Coq, which makes it less practical to reason about large programs in bitVMs.

Bug finders for cryptography programs. Writing correct yet efficient cryptography programs requires specialized domain expertise. A Static analyzer called Circomspect [9] was designed to find bugs in Circom programs. Circomspect looks for simple syntactic patterns such as using the <-- operator when <== can be used. Such a syntactic pattern-matching approach generates many false positives and can also miss real bugs. In contrast, Zkap [35] significantly improves the prior work by reasoning about semantic violations in zero-knowledge circuits. However, those tools are effective in detecting common vulnerabilities with known patterns and can not detect functional violations in cryptography programs, including the benchmarks in our evaluation.

Constraint solving. Satisfiability Modulo Theories (SMT)[22] has become an essential tool for symbolic reasoning, driven by the availability of practical, high-performance solvers like Z3[10], CVC4[3], and Gurobi[14]. The programming languages community has extensively explored the use of solvers for both verification and synthesis [18, 29, 31]. Traditional SMT-based tools often rely on either custom-built constraint solvers or manual translation of problems into constraints for existing solvers. In contrast, solver-aided domain-specific languages (DSLs)[33, 34] automatically generate these constraints through symbolic compilation. One example is the Rosette framework[33], which leverages Racket's meta-programming capabilities to provide a high-level interface to multiple solvers. Building on top of Rosette, BitGuard employs a specialized compilation strategy in Section 3 to produce highly efficient constraints, resulting in a significant reduction in solving time.

9 Conclusion

We have introduced the first formal verification tool tailored for BitVM implementations, addressing the challenges of Bitcoin's constrained programming environment. By designing a higher-level, register-based domain-specific language (DSL) that abstracts away complex stack operations while preserving the original semantics, we bridge the gap between low-level execution and effective program reasoning. Our formal computational model and the use of loop invariant predicates, combined with a counterexample-guided inductive synthesis (CEGIS) procedure, efficiently handle large programs and complex constraints that standard SMT solvers struggle with.

Our evaluation confirms the practicality and effectiveness of our approach. Applied to 98 benchmarks from BitVM's SNARK verifier, our tool successfully verified 94% of the cases within seconds. These findings underscore the tool's potential to significantly enhance the security and reliability of BitVM and pave the way for more secure blockchain applications built on Bitcoin.

References

[1] J. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on sigma-protocols. volume 6345, pages 151–167, 09 2010.

[2] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.

[3] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Annual Design Automation Conference*, DAC '98, page 522–527, New York, NY, USA, 1998. Association for Computing Machinery.

[5] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub. Easycrypt: A tutorial. In *FOSAD*, 2013.

[6] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.

[7] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In H. Boehm and C. Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 3–14. ACM, 2013.

[8] R. Corin and J. den Hartog. A probabilistic hoare-style logic for game-based cryptographic proofs (extended version), 2005. To appear in ICALP 2006 Track C corin@cs.utwente.nl 13264 received 23 Dec 2005, last revised 26 Apr 2006.

[9] F. Dahlgren. It pays to be circomspect. https://blog.trailofbits.com/2022/09/15/it-pays-to-be-circomspect/, 09 2022.

[10] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, 2008.

[11] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, page 500–517, Berlin, Heidelberg, 2001. Springer-Verlag.

[12] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019.

[13] M. Gagné, P. Lafourcade, and Y. Lakhnech. Automated security proofs for almost-universal hash for mac verification. Cryptology ePrint Archive, Paper 2013/407, 2013. https://eprint.iacr.org/2013/407.

[14] L. Gurobi Optimization. Gurobi optimizer reference manual, 2019.

[15] T. Hader. Non-linear smt-reasoning over finite fields, 2022.

[16] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[18] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[19] R. Linus, L. Aumayr, A. Zamyatin, A. Pelosi, Z. Avarikioti, and M. Maffei. BitVM2: Bridging bitcoin to second layers, Aug. 2024.

[20] J. Liu, I. Kretz, H. Liu, B. Tan, J. Wang, Y. Sun, L. Pearson, A. Miltner, I. Dillig, and Y. Feng. Certifying zero-knowledge circuits with refinement types. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 1741–1759. IEEE, 2024.

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

[22] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, Apr. 1980.

[23] A. Niemetz and M. Preiner. Bitwuzla. In C. Enea and A. Lal, editors, *Computer Aided Verification*, pages 3–17, Cham, 2023. Springer Nature Switzerland.

[24] T. Nipkow, M. Wenzel, and L. C. Paulson. Isabelle/hol: A proof assistant for higher-order logic. 2002.

[25] A. Ozdemir. Cvc5-ff. https://github.com/alex-ozdemir/CVC4/tree/ff, 2022.

[26] A. Ozdemir, G. Kremer, C. Tinelli, and C. W. Barrett. Satisfiability modulo finite fields. In C. Enea and A. Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 163–186. Springer, 2023.

[27] S. Pailoor, Y. Chen, F. Wang, C. Rodríguez, J. V. Gaffen, J. Morton, M. Chu, B. Gu, Y. Feng, and I. Dillig. Automated detection of underconstrained circuits for zero-knowledge proofs. Cryptology ePrint Archive, Paper 2023/512, 2023. https://eprint.iacr.org/2023/512.

[28] Polygon. Scalable payments, with zero-knowledge rollups. https://hermez.io, 2022.

[29] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 305–316,

New York, NY, USA, 2013. ACM.

[30] N. Sidorenco, S. Oechsner, and B. Spitters. Formal security analysis of mpc-in-the-head zero-knowledge protocols. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*, pages 1–14, 2021.

[31] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.

[32] A. Tiwari, A. Gascon, and B. Dutertre. Program synthesis using dual interpretation. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *LNCS*, pages 482–497, 2015.

[33] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 530–541, New York, NY, USA, 2014. Association for Computing Machinery.

[34] R. Uhler and N. Dave. Smten with satisfiability-based search. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 157–176, New York, NY, USA, 2014. ACM.

[35] H. Wen, J. Stephens, Y. Chen, K. Ferles, S. Pailoor, K. Charbonnet, I. Dillig, and Y. Feng. Practical security analysis of zero-knowledge proof circuits. In D. Balzarotti and W. Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

$$\frac{z}{\Vdash z}\ (\textsc{Default}) \qquad \frac{\Vdash \mathsf{OP\_?v}}{\Vdash \widetilde{\mathsf{OP}}(v)}\ (\textsc{Const}) \qquad \frac{\Vdash \mathsf{OP\_FALSE}}{\Vdash \widetilde{\mathsf{OP}}(0)}\ (\textsc{False}) \qquad \frac{\Vdash \mathsf{OP\_TRUE}}{\Vdash \widetilde{\mathsf{OP}}(1)}\ (\textsc{True}) \qquad \frac{\Vdash \mathsf{OP\_1NEGATE}}{\Vdash \widetilde{\mathsf{OP}}(-1)}\ (\textsc{Negate})$$

$$\frac{\begin{array}{c}\Vdash \mathsf{OP\_PUSHBYTES\_?n}, z \\ X = \mathrm{asBytes}(z) \qquad v = \Sigma_{i=0:n-1} X[i] \cdot 256^{i}\end{array}}{\Vdash \widetilde{\mathsf{OP}}(v)}\ (\textsc{PushBytes}) \qquad \frac{\Vdash \mathsf{OP\_IF}, z_0, \ldots, z_n, \mathsf{OP\_ENDIF} \qquad \vec{z} = (z_0, \ldots, z_n)}{\Vdash \widetilde{\mathsf{IF}}(\vec{z})}\ (\textsc{If})$$

$$\frac{\begin{array}{c}\Vdash \mathsf{OP\_IF}, z_0, \ldots, z_n, \mathsf{OP\_ELSE}, z_{n+1}, \ldots, z_m, \mathsf{OP\_ENDIF} \\ \vec{z}_a = (z_0, \ldots, z_n) \qquad \vec{z}_b = (z_{n+1}, \ldots, z_m)\end{array}}{\Vdash \widetilde{\mathsf{IFELS}}(\vec{z}_a, \vec{z}_b)}\ (\textsc{IfEls})$$

Fig. 10. Bitcoin script parsing rules.

$$\frac{z \equiv \widetilde{\mathsf{OP}}(v) \qquad \alpha' = \overline{\mathrm{push}}(\alpha, v)}{\langle z, \gamma, \alpha, \beta, \pi \rangle \hookrightarrow \langle \varnothing, \gamma, \alpha', \beta, \pi \rangle}\ (\text{OP-}\textsc{Const}) \qquad \frac{z \equiv \mathsf{OP\_NOP}}{\langle z, \gamma, \alpha, \beta, \pi \rangle \hookrightarrow \langle \varnothing, \gamma, \alpha, \beta, \pi \rangle}\ (\text{OP-NOP})$$

$$\frac{\begin{array}{c}\vec{z} = (z_0, \ldots, z_n) \qquad \langle z_0, \gamma, \alpha, \beta, \pi \wedge c \rangle \hookrightarrow \langle \varnothing, \gamma_0, \alpha_0, \beta_0, \pi_0 \rangle \qquad \ldots \\ \langle z_n, \gamma_{n-1}, \alpha_{n-1}, \beta_{n-1}, \pi_{n-1} \rangle \hookrightarrow \langle \varnothing, \gamma_n, \alpha_n, \beta_n, \pi_n \rangle\end{array}}{\langle \vec{z}, \gamma, \alpha, \beta, \pi \rangle \hookrightarrow \langle \varnothing, \gamma_n, \alpha_n, \beta_n, \pi_n \rangle}\ (\textsc{Batch})$$

$$\frac{\begin{array}{c}z \equiv \widetilde{IF}(\vec{z}) \qquad \alpha_c, c = \overline{\mathrm{pop}}(\alpha) \\ \langle \vec{z}, \gamma, \alpha_c, \beta, \pi \wedge c \rangle \hookrightarrow \langle \varnothing, \gamma_c, \alpha_c, \beta_c, \pi_c \rangle \\ \gamma' = \gamma \cup \gamma_c \qquad \alpha' = \alpha \uplus \alpha_c \qquad \beta' = \beta \uplus \beta_c\end{array}}{\langle z, \gamma, \alpha, \beta, \pi \rangle \hookrightarrow \langle \varnothing, \gamma', \alpha', \beta', \pi \rangle}\ (\text{IF})$$

$$\frac{\begin{array}{c}z \equiv \widetilde{\mathsf{IFELS}}(\vec{z}_a, \vec{z}_b) \qquad \alpha_c, c = \overline{\mathrm{pop}}(\alpha) \qquad \langle \vec{z}_a, \gamma, \alpha_c, \beta, \pi \wedge c \rangle \hookrightarrow \langle \varnothing, \gamma_a, \alpha_a, \beta_a, \pi_a \rangle \\ \langle \vec{z}_b, \gamma, \alpha_c, \beta, \pi \wedge \neg c \rangle \hookrightarrow \langle \varnothing, \gamma_b, \alpha_b, \beta_b, \pi_b \rangle \qquad \gamma' = \gamma \cup \gamma_a \cup \gamma_b \qquad \alpha' = \alpha \uplus \alpha_a \uplus \alpha_b \qquad \beta' = \beta \uplus \beta_a \uplus \beta_b\end{array}}{\langle z, \gamma, \alpha, \beta, \pi \rangle \hookrightarrow \langle \varnothing, \gamma_n, \alpha_n, \beta_n, \pi \rangle}\ (\text{IFELS})$$

$$\frac{\alpha', c = \overline{\mathrm{pop}}(\alpha) \qquad \beta' = \overline{\mathrm{push}}(\beta, c)}{\langle z, \gamma, \alpha, \beta, \pi \rangle \hookrightarrow \langle \varnothing, \gamma, \alpha', \beta', \pi \rangle}\ (\text{OP-TOALTSTACK}) \qquad \frac{\beta', c = \overline{\mathrm{pop}}(\beta) \qquad \alpha' = \overline{\mathrm{push}}(\alpha, c)}{\langle z, \gamma, \alpha, \beta, \pi \rangle \hookrightarrow \langle \varnothing, \gamma, \alpha', \beta', \pi \rangle}\ (\text{OP-FROMALTSTACK})$$

Fig. 11. A subset of the Bitcoin script symbolic evaluation rules.

A Symbolic Evaluation for the Bitcoin Script Language