HANZHI LIU, Nubit and University of California, Santa Barbara JINGYU KE, Nubit HONGBO WEN, Nubit and University of California, Santa Barbara LUKE PEARSON, Polychain Capital 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 verifier 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 low-level program. 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 78 benchmarks from BitVM implementations, our tool successfully verifies 83% of cases within 12.55 seconds on average and identified one previously unknown vulnerability, 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 [23], 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

Authors' Contact Information: Hanzhi Liu, hanzhi@ucsb.edu, Nubit and University of California, Santa Barbara; Jingyu Ke, windocotber@riema.xyz, Nubit; Hongbo Wen, hongbowen@ucsb.edu, Nubit and University of California, Santa Barbara; Luke Pearson, luke@polychain.capital, Polychain Capital; 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; Yu Feng, yufeng@ucsb.edu, Nubit and University of California, Santa Barbara; Yu Feng, yufeng@ucsb.edu, Nubit and University of California, Santa Barbara; Yu Feng, yufeng@ucsb.edu, Nubit and University of California, Santa Barbara; Yu Feng, yufeng@ucsb.edu, Nubit and University of California, Santa Barbara; Yu Feng, yufeng@ucsb.edu, Nubit and University of California, Santa Barbara.

ecosystem. Recently, there has been a significant surge in interest in expanding Bitcoin's capabilities by building ecosystem projects such as Bitcoin Decentralized Finance (BTCFi), aiming to introduce smart contracts and decentralized financial services directly onto the Bitcoin network. However, Bitcoin's inherent scalability and throughput limitations—processing approximately four transactions per second—pose substantial challenges to such developments.

To address these limitations, and inspired by the success of Layer 2 (L2) solutions [30] in scaling Ethereum [8], a promising approach is to leverage recent advancements in BitVM [21], which enables complex computations to be executed off-chain while ensuring their correctness through on-chain cryptography proof verification [6]. 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 [8], which 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 [7] 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 [6] that requires only 200 lines of Solidity code on Ethereum could result in a Bitcoin script program with several gigabytes. 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, even small mistakes 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 BitVM through formal verification becomes imperative and paramount, as these issues can undermine the integrity of the entire system.

However, 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 [12, 27] to handle.

Our key insight is based on two observations. First, although the low-level stack operations are hard to reason about, many of them can be lifted to a higher-level domain-specific language (DSL) that simplifies the reasoning process by replacing intricate stack manipulations with more straightforward register-based operations and concise higher-order functions. 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, **bitguarð**, is motivated by recent successes in program lifting and synthesis [9]. First, we design a register-based, slightly higher-level DSL **6** 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 [33] 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 [18], 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 78 benchmarks derived from various implementations of BitVM. Our tool successfully verified 83% of the cases, demonstrating both its effectiveness and practicality. The verification process is efficient, with an average runtime of 12.55 seconds per benchmark. Finally, bitguard identified one previously unknown bug and our ablation study also demonstrates the benefit of our synthesis approach, especially on complex benchmarks.

In summary, our contributions are as follows:

- We propose the first tool that facilitates formal verification of BitVM implementations, enabling developers to specify and verify correctness properties effectively.
- We design a higher-level DSL 65 that abstracts away complex stack operations and allows for efficient reasoning about BitVM programs while preserving the semantics of the original program.
- We identify and utilize loop invariants to summarize repetitive "loop-style" computations, reducing the complexity of the generated SMT formulas.
- 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.
- Through extensive evaluation, we demonstrate the tool's scalability and its capability to uncover critical vulnerabilities, contributing to the overall security of BitVM.

# 2 Background

*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 [23], the first and most well-known cryptocurrency, was introduced in 2008 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 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 2024, Bitcoin handles around 350,000 daily transactions, with a market cap exceeding \$1.5 trillion, 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 [8].

*Scaling Bitcoin through BitVM*. Bitcoin's original design comes with significant limitations in terms of throughput and scalability. With an average block size of 4 MB and a block time of roughly 10 minutes, Bitcoin can handle only about four transactions per second (TPS). This limited throughput, combined with high transaction fees (which can spike during periods of network congestion), makes Bitcoin less suitable for complex decentralized applications and financial use cases. To address this, BitVM [21] was introduced in 2023 to bring complex programming 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 workload without sacrificing 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 Rust, need to be rewritten or optimized to fit Bitcoin's limited capabilities and resource constraints.

**Bugs in BitVM implementations.** The complexity of compiling high-level application 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, potentially leading to loss of funds.

Applying formal verification to BitVM presents several significant challenges. First, the pervasive use of low-level stack operations intertwined with application logic creates complex and errorprone reasoning paths. Second, the ubiquity of non-linear arithmetic, a notoriously difficult area for formal methods, further complicates verification. Third, the limited expressiveness of Bitcoin Script leads to verbose and expansive codebases, making the analysis both time-consuming and resource-intensive. These factors combine to make formal verification of BitVM both technically demanding and computationally costly.

#### 3 Overview

In this section, we motivate our proposed approach, bitguard, with a motivating example.

*Motivating example.* The left-hand side of Figure 1 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.

*A 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 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 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.



Fig. 1. A motivating example showing a partial Bitcoin script for computing big integer multiplication and its corresponding  $\mathfrak{G}$  program synthesized by bitguard. One then checks the correctness of the snippet by providing specification written in  $\mathfrak{G}$ , which produces optimized constraints.



Fig. 2. Illustration of the operations within a single loop iteration of big integer multiplication.

This process is repeated for all bits in the multiplier's limbs, and the results are combined to form the final product.

*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 above-mentioned challenges, 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.
- Lifting Loop Patterns to Higher-order Functions 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 using *higher-order functions*. 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 2, 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 synthesis [33], our key insight is to synthesize and lift the Bitcoin script into its equivalent high-level representation. The program in the middle of Figure 1 shows the equivalent version in our DSL <sup>6</sup>. 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 higher-order functions such as map.

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. 3. A high-level overview of the bitguard verification framework.

- **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:  $A = \text{BigInt}(A_0) \land R = 0$ .
- **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:  $R' = R + B[i] \times 2A \wedge A' = 2A$ . As shown in Figure 2, this loop invariant captures the relationship between the intermediate result *R*, the *i*-th bit B[i] from the multiplier, and the multiplicand *A* after the *i*-th iteration.
- 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:

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

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 by an off-the-shelf SMT solver [2, 12], 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 tackle more complex cryptographic operations like BigInt.

# 4 The Verification Algorithm

In this section, we introduce the overall verification algorithm of **bitguard**. We first describe a high-level overview of the system, including its key procedures. Then we introduce the domain-specific language **G** built within **bitguard**, which can be used to summarize stack-based operations in a verification-friendly way. As an improvement to verification, **G** can be further strengthened by user-provided specification and loop invariants.

# 4.1 System Overview

As shown in Figure 3, bitguard takes inputs as a Bitcoin script that implements a full system such as BitVM [21] and user-provided specifications. It then outputs whether the given system is safe regarding the specification, in particular, in three potential outcomes: safe ( $\checkmark$ ), unsafe ( $\times$ ) or unknown (?). Specifically, bitguard contains two major phases:

- Transpilation As stack-based computations from the original system are usually difficult to reason about, bitguard addresses this problem by *synthesizing* its equivalent version that is easier to verify. In particular, bitguard first decomposes the original system into independent code snippets, and *rewrites* each of them into its equivalent snippet in the  $\mathfrak{G}$  language (i.e.,  $\mathfrak{G}$  snippet) via program synthesis. The  $\mathfrak{G}$  language of bitguard provides a verification-friendly interface that makes it easy and efficient for reasoning about the system's behavior. Then by assembly of the  $\mathfrak{G}$  snippets, bitguard gets a  $\mathfrak{G}$  program that is optimized for verification.
- Verification The user then provides a specification (e.g., precondition, postcondition, and verification condition) within the synthesized **G** program using **G**'s verification interface (e.g., assume and assert). For loops in the program, **bitguard** performs a Houdini-style algorithm for loop invariant inference, which rewrites and summarizes loops in a more concise form. Thus, in the *verification* phase, **bitguard** devises a set of symbolic evaluation rules that convert the given **G** program into a set of logical constraints that an off-the-shelf solver can reason about.

We first elaborate on the verification procedure and defer a detailed discussion of the transpilation phase to Section 5. Specifically, we give an introduction of the  $\mathfrak{G}$  language in Section 4.2, including its core syntax for modeling program behavior and writing verification queries. Building on top of  $\mathfrak{G}$ , we then describe how an  $\mathfrak{G}$  program can be optimized for verification with an algorithm for automatic inference of loop invariants. Finally, a set of symbolic evaluation and merging rules is introduced in Section 4.3, which convert an  $\mathfrak{G}$  program into logical constraints, thus reducing a verification task into constraint solving.

# 4.2 The 6 Language for Modeling Stack Operations

Figure 4 shows the syntax of our **6** language, for modeling stack-based computations in Bitcoin script. From a high-level perspective, **6** is a functional programming language with *higher-order functions* for batched stack operations and verification. The top level of an **6** program consists of a sequence of statements from three different categories:

- Basic Types and Control Flows There are three basic types in 𝔅, namely booleans, integers and hashes. In addition to standard arithmetic operators for booleans (⊗) and integers (⊕), 𝔅 also models cryptographic operations (☉) such as sha1 and hash160, which compute hashes as their output. 𝔅 models standard control flows such as branches and loops. Note that a loop in 𝔅 by default has a constant bound *c* (i.e., *bounded*) due to the nature of stack-based scripts.
- Stack Operations (5) incorporates higher-order functions that perform stack-based computations in a batched manner without exposing details of low-level data structures. Specifically:
  - The **append** operator pushes to the top of the stack a new set of elements.
  - The **switch** operator moves a subset of stack elements into another stack; e.g., if the specified elements are in the main stack, then they will be moved to the alt stack; vice versa.
  - The **map** operator is a higher-order operator, which selects a subset of stack elements, and applies a function with argument *c* in place to each element in the subset.
  - The **mapsto** operator performs a similar operation as the map operator does, except that mapsto moves the resulting subset of elements to the top of the stack.
  - The filter operator selects a subset of stack elements that satisfy the given condition, and moves the results to the top of the stack.
  - The **fold** operator is a higher-order operator that consumes a seed value *c* and a subset of stack elements and progressively constructs a result on top of the stack with the function •.
  - The zip operator is a higher-order operator, which takes two subsets of stack elements and applies a function to each pair of them. The resulting set of elements is then pushed to the top of the stack.

s e	::=   	s * e; $\sigma;$ i c $\diamond$ $k \equiv b   z   b$	if (e) then p else p. loop (c) p. $i \leftarrow e$ ; assume(e); assert(e); $\mu[c*]$	Statement: sequence expression stack operation branch loop assignment assumption assertion Expression: identifier constant symbolic stack accessor arithmetic expres	$ \begin{array}{c} b \\ z \\ h \\ \sigma \end{array} $ • • • • • • • • • • • • • • • • • •	::= ::=   	$\neg e \mid e \otimes e$ -e \mid e \oplus e $\bigcirc e$ append(c*) switch(k) mapto(k, •, c) filter(k, ⊗, c) map(k, •, c) fold(k, •, c) zip(k, k, •) {⊗, ⊕, \bigcirc, ⊖} {\land, \lor, =, \neq, <, \leq,} {+, -, *, /,}	Boolean Expr. Integer Expr. Hashing Expr. Stack Operation: stack append move bt. stacks stack mapto stack filter stack map stack fold stack zip Operators Boolean Ops. Integer Ops. Hashing Ops
		$b \mid z \mid h$	μ) μ[ο.]	arithmetic exprs.	0	e	{sha1, hash160,}	Hashing Ops.
μ	e	{main, a	11 }	Stack Selectors	θ	e	{mv, cp, natzip, }	Stack Ops.

Fig. 4. A representative set of the syntax of  $\mathfrak{G}$  programs.

• Verification Constructs  $\mathfrak{G}$  incorporates two constructs for verification queries, namely assume and assert, where assume takes a boolean expression e and appends it to the current *path condition* as additional assumption, and assert checks in place whether the given expression eevaluates to true. In a  $\mathfrak{G}$  program, a verification query e can be built from  $\mathfrak{G}$  expressions, and tracked with the assignment construct  $i \leftarrow e$  in a dedicated environment besides the stacks.

*Example 4.1 (A Program in* **6***).* The following shows a **6** program:

map(main[0:3], +, 1); zip(main[0:3], main[3:6], \*);

which first adds 1 to the first three elements, and then multiplies each pair of the first three and second three elements. The results are pushed to the top of the stack.

*Inference of Loop Invariants.* For a loop statement, we implement a Houdini-style [13] inference algorithm that generates conjunctive invariants. This baseline generates all possible atomic predicates by unwinding the grammar that captures common templates in our domain up to a fixed bound and generates the strongest conjunctive invariant over this universe in the standard way.

# 4.3 Symbolic Evaluation for the 6 Language

We then describe how **bitguard** symbolically evaluates a  $\mathfrak{G}$  program and keeps track of program states via a set of evaluation rules. We refer to a *program state* as a 4-tuple  $\langle p, \gamma, \delta, \pi \rangle$ , where:

- *p* is the *program counter* that points to the immediate next **(5)** statement.
- $\gamma$  is the *assertion store* that tracks verification conditions generated during symbolic evaluation, which can be implied by  $\mathfrak{G}$  language constructs or derived from user-provided specification.
- $\delta$  is the *program store* that provides access to the memory and stacks. Specially, a stack operation  $\sigma$  can access both the main and alt stacks by the form  $\delta$ [main] and  $\delta$ [alt]; besides, the verification interface can access the memory with given identifier *i*, in the form  $\delta$ [*i*].
- $\pi$  keeps track of the current *path condition*, which is a boolean value that evaluates to true in the current program state, and remains true in order to reach the next program state; otherwise, the next program state is said to be *unreachable*.

During transition of program states, if a value *x* can only be accessed under certain path condition  $\pi$ , we then say *x* is *guarded* by  $\pi$ , denoted by  $(|\pi|)x$ . Thus, each slot *i* of the program store  $\delta$ , also denoted as  $\delta[i]$ , is mapped to a set of possible values guarded by different path conditions:

$$\delta[i] = \{ (\pi_0) x_0, ..., (\pi_n) x_n \}.$$

Liu et al.

$$\frac{\langle \mathfrak{s}_{0}, \mathfrak{r}, \delta, \pi \rangle \rightsquigarrow \langle \mathfrak{O}, \mathfrak{r}_{0}, \delta_{0}, \pi_{0} \rangle \qquad \dots \qquad \langle \mathfrak{s}_{n}, \mathfrak{r}_{n-1}, \delta_{n-1}, \pi_{n-1} \rangle \rightsquigarrow \langle \mathfrak{O}, \mathfrak{r}_{n}, \delta_{n}, \pi_{n} \rangle}{\langle \mathfrak{s}_{0}, \mathfrak{r}_{0}, \mathfrak{s}_{0}, \mathfrak{r}_{0} \rangle} (SEQN)$$

$$\frac{\langle \mathfrak{s}_{0}, \mathfrak{r}, \delta, \pi \rangle \rightsquigarrow \langle \mathfrak{o}, \mathfrak{r}, \delta, \pi \rangle}{\langle (\mathfrak{s}_{0}, \dots, \mathfrak{s}_{n}), \mathfrak{r}, \delta, \pi \rangle \rightsquigarrow \langle \mathfrak{O}, \mathfrak{r}_{n}, \delta_{n}, \pi_{n} \rangle} (SEQN)$$

$$\frac{\langle \mathfrak{s}_{0}, \mathfrak{r}, \delta, \pi \rangle \rightsquigarrow \langle \mathfrak{o}, \mathfrak{r}, \delta, \pi \rangle \rightsquigarrow \langle \mathfrak{o}, \mathfrak{r}, \delta, \pi \rangle \rightsquigarrow \langle \mathfrak{O}, \mathfrak{r}, \delta, \pi \rangle}{\langle \mathfrak{o}, \mathfrak{r}, \delta, \pi \rangle \rightsquigarrow \langle \mathfrak{o}, \mathfrak{r}, \delta, \pi \rangle \rightsquigarrow \langle \mathfrak{O}, \mathfrak{r}, \delta, \pi \rangle} (SYME) \qquad \overline{\langle (\mathfrak{i}, \mathfrak{r}, \delta, \pi \rangle \rightsquigarrow \langle \mathfrak{O}, \mathfrak{r}, \delta, \pi \rangle} (IDEN)$$

$$\frac{\langle \mathfrak{e}, \mathfrak{r}, \delta, \pi \rangle \rightsquigarrow \langle \mathfrak{o}, \mathfrak{r}_{e}, \delta_{e}, \pi_{e}, \wedge \mathfrak{o} \rangle \rightsquigarrow \langle \mathfrak{O}, \mathfrak{r}, \delta, \pi_{e} \rangle}{\langle \mathfrak{p}_{0}, \mathfrak{r}_{e}, \delta_{e}, \pi_{e}, \wedge \mathfrak{o} \rangle \rightsquigarrow \langle \mathfrak{O}, \mathfrak{r}, \delta, \pi_{0} \rangle} \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{o}, \pi_{0} \rangle \langle \mathfrak{O}, \mathfrak{r}, \delta, \pi_{0} \rangle \sim \langle \mathfrak{O}, \mathfrak{r}, \delta, \pi_{0} \rangle} \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{o}, \mathfrak{r}, \mathfrak{o} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{o}, \pi_{0} \rangle} \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{o}, \mathfrak{r}, \mathfrak{o} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{o}, \mathfrak{r}, \mathfrak{o} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{o}, \mathfrak{r} \rangle}{\langle \mathfrak{o}, \mathfrak{r}, \mathfrak{o}, \pi_{0} \rangle \rightsquigarrow \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{o}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{o}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{o}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{O}, \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle \langle \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{r}, \mathfrak{r}, \mathfrak{r} \rangle} \langle \mathfrak{r}, \mathfrak{r}, \mathfrak{r}$$

Fig. 5. A representative set of the symbolic evaluation rules (part 1) for the control flow constructs and verification interface of  $\mathfrak{G}$ .

Consider accessing a given slot *i* in the program store  $\delta$ , only those values guarded by  $\pi'$  which *implies* the current path condition  $\pi$  can be successfully retrieved; we use the form  $\delta_{\pi}[\![i]\!]$  (or  $\delta[\![i]\!]$  for short) to denote access to program store  $\pi$  under path condition  $\pi$ :

$$\delta\llbracket i\rrbracket = \delta_{\pi}\llbracket i\rrbracket = \{ (\pi') | x \in \delta[i] \mid \pi' \Longrightarrow \pi \}.$$

*Symbolic evaluation rules*. Figure 5 shows a representative set of symbolic evaluation rules for the control flow constructs and verification interface of **(6)**. The following judgment:

$$\langle p, \gamma, \delta, \pi \rangle \rightsquigarrow \langle q, \gamma', \delta', \pi' \rangle$$

denotes a successful execution of the form *p* in the program state  $\langle p, \gamma, \delta, \pi \rangle$  and results in the return form *q* in the program state  $\langle q, \gamma', \delta', \pi' \rangle$ .

The evaluation process starts with the (SEQN) rule, which populates each statement *s* within the given sequence ( $s_0$ , ...,  $s_n$ ) and evaluates them accordingly. Rules (CNST), (SYMB), and (IDEN) define three different ways to retrieve data via directly providing constant value, symbolic value, and access to the program store  $\delta$ . Note that each constant or symbolic value is typed; it's either a boolean, integer or hash. Thus, binary expression (BEXP) and unary expression (UEXP) require operands to match the type requirement of the corresponding operators.

The (BNCH) rule denotes how a program state should be tracked for separate execution branches, and merged afterward: The condition e will first be evaluated and the resulting condition v is then conjoined with the current path condition  $\pi$  for evaluation of the then-branch; for the else-branch, the negation of the condition  $\neg v$  is conjoined instead. The two ending program states are then merged. In particular, assertion stores are merged by disjunction, and program stores are merged per each value mapping. Given program stores  $\delta_0$  and  $\delta_1$ , their merged version  $\delta_0 \uplus \delta_1$  is given by:

$$\delta_{0} \uplus \delta_{1} = A \cup B \cup C,$$
  
where  $A = \{i \mapsto \delta_{0}[i] \mid i \in \operatorname{dom}(\delta_{0}) \setminus \operatorname{dom}(\delta_{1})\}, B = \{i \mapsto \delta_{1}[i] \mid i \in \operatorname{dom}(\delta_{1}) \setminus \operatorname{dom}(\delta_{0})\},$   
and  $C = \{i \mapsto \delta_{0}[i] \cup \delta_{1}[i] \mid i \in \operatorname{dom}(\delta_{0}) \cap \operatorname{dom}(\delta_{1})\}.$ 

$$\frac{l = \{c_0, ..., c_n\} \quad X = l \cup \delta[[\text{main}]] \quad \delta' = \delta \cup \{\text{main} \mapsto X\}}{\langle \text{append}(l), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \emptyset, \gamma, \delta', \pi \rangle} \quad (\text{Append})$$

$$\begin{split} \mu' &= \text{alt if } \mu \equiv \text{main else main} \qquad M = \delta[\![\mu]\!] \qquad M' = \delta[\![\mu']\!] \\ A &= \delta[\![\mu]\!] [\![l]\!] \qquad X_0 = M \backslash A \qquad X_1 = A \cup M' \\ \hline \delta' &= \delta \cup \{\mu \mapsto X_0, \mu' \mapsto X_1\} \qquad \gamma' = \gamma \cup \gamma_0 \\ \hline \langle \text{switch}(\mu[l]), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \emptyset, \gamma', \delta', \pi \rangle \end{split}$$
(Switch)

$$\begin{split} & M = \delta[\![\mu]\!] \quad A = \delta[\![\mu]\!] [l]\!] \quad A' = \text{flatten}(\{v \bullet c \mid v \in A\}) \\ & \frac{X = A' \cup (M \setminus A) \quad \delta' = \delta \cup \{\mu \mapsto X\} \quad \gamma' = \gamma \cup \gamma_a}{\langle \text{mapto}(\mu[l], \bullet, c), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \emptyset, \gamma', \delta', \pi \rangle} \quad \text{(Mapto)} \end{split}$$

$$\begin{array}{l}
M = \delta[\![\mu]\!] & A = \delta[\![\mu]\!][\![l]\!] & A' = \{v \mid v \otimes c = \operatorname{true} \land v \in A\} \\
\frac{X = A' \cup (M \setminus A)}{\langle \operatorname{filter}(\mu[l], \otimes, c), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \emptyset, \gamma', \delta', \pi \rangle} & (\operatorname{Filter})
\end{array}$$

$$\frac{M = \delta[\![\mu]\!] \qquad X = \{M[\![i]\!] \bullet c \text{ if } i \in l \text{ else } M[\![i]\!] \mid 0 \le i \le |M|\}}{\delta' = \delta \cup \{\mu \mapsto X\}} \qquad \gamma' = \gamma \cup \gamma_0} \qquad (Map)$$

$$\begin{split} l &= \{c_0, ..., c_n\} \qquad M = \delta[\![\mu]\!] \qquad A = \delta[\![\mu]\!] [\![l]\!] \qquad v_0 = c \bullet M[\![c_0]\!] \\ &\dots \qquad v_n = v_{n-1} \bullet M[\![c_n]\!] \qquad X = \{v_n\} \cup (M \backslash A) \\ &\frac{\delta' = \delta \cup \{\mu \mapsto X\} \qquad \gamma' = \gamma \cup \gamma_a}{\langle \text{fold}(\mu[l], \bullet, c), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \emptyset, \gamma', \delta', \pi \rangle} \end{split} \tag{Fold}$$

$$\begin{split} l_{a} &= \{c_{0}, ..., c_{n}\} \qquad l_{b} = \{d_{0}, ..., d_{n}\} \qquad M = \delta[\![\mu]\!] \qquad A = \delta[\![\mu]\!][\![l_{a}]\!] \\ B &= \delta[\![\mu]\!][\![l_{b}]\!] \qquad X = \{A[\![c_{i}]\!] \bullet B[\![d_{i}]\!] \mid 0 \le i \le n\} \cup (M \backslash A \backslash B) \\ \\ \hline \delta' &= \delta \cup \{\mu \mapsto X\} \qquad \gamma' = \gamma \cup \gamma_{b} \\ \hline \langle \text{zipwith}(\mu[l_{a}], \mu[l_{b}], \bullet), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \emptyset, \gamma', \delta', \pi \rangle \qquad (\text{ZIP}) \end{split}$$















Fig. 6. A representative set of the symbolic evaluation rules (part 2) for the stack operations of  $\mathfrak{G}$ . We illustrate changes to stacks before and after the corresponding stack operation to the right of each rule.

Here, dom( $\delta$ ) denotes the set of identifiers in the program store  $\delta$ .

The (ASGN), (ASUM), and (ASRT) rules denote how the verification interface interacts with the program state. The assignment rule (ASGN) binds a location in the program store  $\delta$  to an identifier *i*. The assumption rule (ASUM) adds the resulting value *v* of evaluation of the expression *e* into the current path condition  $\pi$  by conjunction. Similarly, the assertion rule (ASRT) appends *v* to the assertion store  $\gamma$ . During the evaluation, bitguard terminates when the current path condition  $\pi$  evaluates to false, or the conjunction of all clauses from the assertion store  $\gamma$  can not be satisfied.

The symbolic evaluation rules of the higher-order constructs for modeling batched stack operations of  $\mathfrak{G}$  are shown in Figure 6. To provide a high-level intuition, each rule is accompanied by a visualization that depicts the stack's state before and after the corresponding operation is applied, highlighting its side effects. Specifically, the (APPEND) and (SWITCH) rules do not change the values of the input elements. The (APPEND) rule moves the input elements to the top of the main stack, while the (SWITCH) rule moves the selected elements between main and alt stacks. The rules for

## Algorithm 1 Synthesis-Powered Transpilation

1	$T_{-}$	
1:	<b>procedure</b> TRANSPILE( $\mathfrak{G}, P$ )	
2:	input: domain-specific language 66, decompo	osed Bitcoin snippet P
3:	<b>output:</b> transpiled $\mathfrak{G}$ program $P'$ or $\perp$ if not	found
4:	$S_{\phi} \leftarrow \text{SEval}(P)$	▷ symbolically evaluates original snippet into logical specification
5:	sample $E \sim \{(e_{in}, e_{out}) \mid P(e_{in}) = e_{out}\}$	▷ samples input-output examples from snippet P
6:	$\kappa \leftarrow \top$	⊳ initializes knowledge base
7:	while $P' \leftarrow \text{Enumerate}(\mathfrak{G}, E, \kappa)$ do	
8:	$\mathcal{S}' \leftarrow \operatorname{SEval}(P')$	symbolically evaluates candidate program into logical constraints
9:	$r \leftarrow \operatorname{sat}(\mathcal{S}' \not\models \mathcal{S}_{\phi})$	<ul> <li>check for counterexample</li> </ul>
10:	if r then	
11:	$(e'_{in}, e'_{out}) \leftarrow cex(r), E \leftarrow E \cup (e'_{in}, e'_{out})$	ut) ▶ gets the counterexample and adds to example set
12:	$\kappa \leftarrow \kappa \land \operatorname{block}(P')$	blocks the current candidate program
13:	else return P'	▹ no counterexample is found; returns the program
14:	return ⊥	⊳ exhausted

the remaining four higher-order operations, namely the rules of (MAPTO), (FILTER), (MAP), (FOLD) and (ZIP), accept an operator • that is used to transform the input elements into new ones. Since the selected elements could be guarded by path conditions, the result of applying an operator • on the guarded values  $g_0$  and  $g_1$  is given by:

$$g_0 \bullet g_1 = (\pi_0 \wedge \pi_1)(x_0 \bullet x_1),$$

where  $g_0 = (\pi_0) x_0$  and  $g_1 = (\pi_1) x_1$ .

## 5 Synthesis-Powered Transpilation

In this section, we introduce the transpilation algorithm that converts a decomposed Bitcoin snippet into its equivalent  $\mathfrak{G}$  program via a counterexample-guided inductive synthesis (CEGIS) loop. The synthesized  $\mathfrak{G}$  program will then be used for reasoning in the verification phase as mentioned in Section 4. We first give an overview of the synthesis algorithm in Section 5.1, and explain in detail the synthesis procedure (Section 5.2) and the equivalence checking (Section 5.3).

## 5.1 Algorithm Overview

As shown in Algorithm 1, given the domain-specific language  $\mathfrak{G}$  and a decomposed Bitcoin script P, bitguard starts by obtaining the synthesis specification  $S_{\phi}$  via symbolic evaluation (i.e., the SEVAL procedure) of P (line 4). It then samples an initial set E of input-output examples from the original snippet P (line 5). Each example  $(e_{in}, e_{out})$  consists of an input  $e_{in}$  and an output  $e_{out}$  that correspond to the status of the stacks before and after applying the snippet P respectively, i.e.,  $P(e_{in}) = e_{out}$ . bitguard then continuously constructs candidate  $\mathfrak{G}$  programs via the ENUMERATE procedure (line 7-14) until a solution is found. Specifically for each proposed candidate program P', bitguard obtains its representation S' in constraints (line 8) and checks if there exists an input-output example (i.e., a counterexample) from S' that violates the synthesis specification  $S_{\phi}$  (line 9). The candidate program P' is not the solution if such a counterexample  $(e'_{in}, e'_{out})$  exists (line 10). In this case, bitguard retrieves the exact counterexample and adds it to the example set E (line 11) while blocking the program P' (line 12); otherwise, if no counterexample is found, the candidate program P' is then returned since it proves to be equivalent to the original snippet P (line 13).

## 5.2 The Enumeration Procedure

Given a domain-specific language, which here refers to  $\mathfrak{G} = (V, \Sigma, R, S)$ , where  $V, \Sigma, R$  and S denote the non-terminals, terminals, productions and start symbol respectively, the enumeration procedure

Operator	Description	Logical Summary
append(x)	pushes new elements to the top of stack	$(\sigma_{m'} = \sigma_m + \sigma_x) \land (\sigma_{a'} = \sigma_a)$
switch(x)	moves elements between stacks	$(\sigma_{m'} = \sigma_m - \sigma_x) \land (\sigma_{a'} = \sigma_a + \sigma_x)$
mapto( <i>x</i> , _, _)	applies a function to each selected elements and	$(\sigma_{m'} \ge \sigma_m) \land (\sigma_{a'} = \sigma_a)$
	moves results to the top of stack	
filter( <i>x</i> , _, _)	selects a subset of elements with conditions and	$(\sigma_{m'} \leq \sigma_m) \land (\sigma_{a'} = \sigma_a)$
	moves results to the top of stack	
map( <i>x</i> , _, _)	applies a function to each selected elements in place	$(\sigma_{m'} = \sigma_m) \land (\sigma_{a'} = \sigma_a)$
fold( <i>x</i> , _, _)	progressively constructs a result to the top of stack	$(\sigma_{m'} = \sigma_m - \sigma_x + 1) \land (\sigma_{a'} = \sigma_a)$
$zip(x_0, x_1, _)$	applies a function to each pair of two sets of elements	$(\sigma_{m'} = \sigma_m - \sigma_{x_0}) \land (\sigma_{a'} = \sigma_a) \land$
	and pushes results to the top of stack	$(\sigma_{x_0} = \sigma_{x_1})$
if $e$ then $p_0$ else $p_1$ .	branch statement	$\phi_{p_0} \lor \phi_{p_1}$
loop (c) p.	loop statement	$\wedge_c \phi_p$

Table 1. A representative set of logical summary of  $\mathfrak{G}$ . *x* denotes the input. *m* and *a* denote the main and alt stack respectively.  $\sigma_p$  denotes the size of *p*, and  $\phi_p$  retrieves the logical summary of *p*. We differentiate a stack's status before and after an operation with ', e.g., *m* (before) and *m*' (after).

finds a *feasible* program *P* in  $\mathfrak{G}$ , such that for all given input-output examples  $(e_{in}, e_{out}) \in E$ , execution of *P* over each input  $e_{in}$  results in the corresponding output  $e_{out}$ .

There are three steps in the enumeration procedure, namely *derivation*, *encoding* and *pruning*. The derivation step constructs a well-typed **(5)** program, which is then encoded with the given input-output examples into a *logical summary*. The enumeration procedure prunes a program if its logical summary proves unsatisfiable and returns it otherwise. We elaborate on the three steps in detail as follows.

**Derivation**. To derive a well-typed program P from  $\mathfrak{G}$  by construction, we model P as a sequence of terminals V and non-terminals  $\Sigma$  in  $\mathfrak{G}$ :  $P \in (V \cup \Sigma)$ \*, such that P can be derived from S via a sequence of productions from R:

$$S \xrightarrow{r*} P$$
 where  $r \in R$ .

A program that contains non-terminals is *partial*, and such non-terminals are also referred to as *holes*. Starting from *S*, by gradually filling in a partial program's holes, the enumeration procedure eventually derives a well-typed and *complete* program without any non-terminals.

*Example 5.1 (Partial Program Derivation).* The following shows a partial program written in  $\mathfrak{G}$ :

mapto( $k_0$ , "mv", 0); zip( $k_1$ ,  $k_2$ , "flat");

where  $k_0$ ,  $k_1$  and  $k_2$  are non-terminals. With the productions  $k ::= \mu[c*], \mu ::=$  main and c ::= 9, we can fill in the hole  $k_0$  and thus derive a new partial program:

mapto(main[9], "mv", 0); 
$$zip(k_1, k_2, "flat")$$
;

**Encoding.** For a given program P, the enumeration procedure performs a *quick* checking of its feasibility over the given set of examples E via its logical summary. We refer to a logical summary as a set of logical formulas that describes the behavior of a language construct in an *abstract* way. For example, Table 1 shows the logical summary for each of the stack operators of  $\mathfrak{G}$ , where x and y denotes the input and output stack of an operator, with certain type of stack specified by subscript (e.g., m for main stack and a for alt stack). Each summary quantifies the relation between the size properties of the input and output stacks. For example, in the logical summary of append, the size of the main stack becomes larger in the output than input but alt stack remains the same; for switch, the main stack shrinks and the alt stack grows.

Thus, let  $\mathfrak{T}_P$  be the AST representation of *P*, we can then encode a program *P* with given input  $e_{in}$  and output  $e_{out}$  into its logical summary  $\Psi(P(e_{in}) = e_{out})$ :

$$\Psi(P(e_{\text{in}}) = e_{\text{out}}) = \bigwedge_{N \in \text{Nodes}(\mathfrak{T}_P)} \phi(N),$$

where  $\phi_n$  denotes the logical summary of the node *n*.

Example 5.2 (Logical Summary). Consider the following partial program:

mapto(main[0:3],  $\bullet_0$ ,  $c_0$ ); zip(main[0:3], main[3:6],  $\bullet_1$ );

Let  $x_0$  be the input of mapto, and  $x_{1a}$ ,  $x_{1b}$  be the inputs of zip. The above program is then encoded to the following logical summary:

$$(\sigma_{m_1} \geq \sigma_{m_0}) \land (\sigma_{a_1} = \sigma_{a_0}) \land (\sigma_{m_2} = \sigma_{m_1} - \sigma_{x_{1a}}) \land (\sigma_{a_2} = \sigma_{a_1}) \land (\sigma_{x_{1a}} = \sigma_{x_{1b}}),$$

where  $m_0$  and  $a_0$  correspond to the initial stacks,  $m_1$  and  $a_1$  are stacks after the first operation mapto,  $m_2$  and  $a_2$  are the final stacks after the second operation zip.

**Pruning**. For each given input-output pair  $(e_{in}, e_{out}) \in E$ , if its logical encoding  $\Psi(P(e_{in}) = e_{out})$  is unsatisfiable, then *P* can be safely pruned. Therefore, the enumeration procedure returns the program *P*, if the following query yields true:

$$\bigwedge_{(e_{in},e_{out})\in E} SAT(\Psi(P(e_{in})=e_{out})).$$

#### 5.3 Equivalence Checking

Once a candidate program P' has been proposed by the enumeration procedure, it is essential to ensure that it is semantically equivalent to the original snippet P. However, verifying this equivalence is non-trivial, as there is no off-the-shelf equivalence checker for comparing Bitcoin script with programs in  $\mathfrak{G}$ . We thus implemented equivalence checking to address this challenge.

The core idea is to symbolically evaluate (via the SEVAL procedure) both programs on a common input state and check if their resulting output states are the same. To build the checker, we adapted existing symbolic evaluation rules for Bitcoin script from existing work [19] with those already defined in Section 4 for **(5)**. The checker was built on top of the rosette framework [35] and leverages its SMT encoding facilities as well as its symbolic evaluation engine.

## 6 Implementation

We have implemented **bitguard** in Racket/Rosette with a back-end constraint solver (Bitwuzla [25] version 0.4.0). The total codebase comprises 2,574 lines of Racket code. 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 signmagnitude 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:

- *N*: The total number of bits of the BigInt.
- *L*: The number of bits per segment (limb).
- I: The base name for each limb, with *l<sub>i</sub>* representing the *i*-th limb.

• v: The identifier for the entire BigInt, with  $v_i$  representing the *i*-th BigInt.

For example, PUSH\_BIGINT\_0 254 29  $s v_0$  creates a 254-bit BitInt, split into limbs of 29 bits each, named  $s_0$ ,  $s_1$  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:

$$v_0 = \sum_{i=0}^n s_i \cdot 2^{\mathcal{L} \cdot i}$$
, where  $n = \left\lceil \frac{N}{\mathcal{L}} \right\rceil - 1$ .

After this operation, the stack will have  $s_0, s_1, ..., s_n$  pushed onto it, where each  $s_i$  is a symbolic bitvector of size  $\mathcal{L}$  (except possibly the highest limb, which may be smaller if  $\mathcal{N}$  is not a multiple of  $\mathcal{L}$ .

*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  $\mathcal{L}$  and constrain it to be within the range  $[0, 2^{\mathcal{L}} - 1]$ .

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

$$\mathcal{L}_h = \mathcal{N} \mod \mathcal{L}.$$

The highest limb is of size  $\mathcal{L}$  if  $\mathcal{L}_h = 0$ .

To ensure that the sign bit is correctly modeled, we constrain the most significant bit of the highest limb to be 0 by default. The position of the sign bit within the highest limb is:

sign = 
$$\begin{cases} \mathcal{L}_h - 1 & \text{if } \mathcal{L}_h > 0, \\ \mathcal{L} - 1 & \text{if } \mathcal{L}_h = 0. \end{cases}$$

We then apply the following constraint to the highest limb  $s_n$ :  $s_n[sign] = 0$ , where  $s_n[i]$  denotes the *i*-th bit of  $s_n$ . 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 nonlinear 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?
- RQ3 (Zero-Days) Is bitguard useful for detecting previously unknown vulnerabilities?

**Benchmarks**. We collect a total of 78 verification tasks from the two major open-source repositories written using Bitcoin script, which contains the usage of a wide coverage of Bitcoin script language constructs in various computational tasks, libraries, and components, as follows:

	Total	Avg. Time	Solved	Safe (🖌)	Unsafe (X)	Unknown (?)
BSE	67	14.43s	54 (81%)	53 (79%)	1 (2%)	13 (19%)
BSV	11	3.49s	11 (100%)	11 (100%)	0 (0%)	0 (0%)
Overall	78	12.55s	65 (83%)	64 (82%)	1 (1%)	13 (17%)

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

- bitcoin-scriptexec<sup>1</sup> (or BSE for short) implements BitVM2 [21], the official implementation from the original authors. It also comes with a library of functions written in Bitcoin script for various computations and operations in arithmetics, cryptography, stack, bitvector, etc.
- **Bitcoin circle STARK verifier**<sup>2</sup>(or **BSV** for short) implements a circle plonk [14] verifier in Bitcoin script. It also comes with reusable cryptographic components written in Bitcoin script.

Among our 78 benchmarks, 67 benchmarks are from BSE and 11 from BSV. Each benchmark has on average 269,739 lines of code, with a maximum of 5,780,711 lines. The computations implemented in the benchmarks mainly fall 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*. All experiments are conducted on a system with an AMD Ryzen 9 5950X 16-Core Processor and 64 GB of memory, running Ubuntu 20.04. *bitguard* encodes semantics of bitcoin script in bitvector theory [4] and leverages Bitwuzla [25] as its default backend constraint solver. The default timeout for evaluation of each benchmark is set to 15 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 ("**✓**"), meaning that the program conforms with the specification;
  - Unsafe ("X"), meaning that a counterexample that violates the specification is found;
  - Unknown (denoted by "?"), meaning that bitguard cannot terminate within a 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 the refinement procedure mentioned in Section 4.2 that validates the counterexample proposed by the tool, due to the fact that the initial loop invariant might not be strong enough to imply the desired post-condition. Specifically, **bitguard** iteratively strengthens the candidate invariant and continues with the verification process until a definite conclusion is reached.

• 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 78 benchmarks, **bitguard** solves 65 (83%) of them, with 64 (82%) of them proven safe ( $\checkmark$ ) and 1 (1%) of them having counterexamples found, i.e., proven unsafe ( $\times$ ). **bitguard** takes an average of 12.55s to solve a

<sup>&</sup>lt;sup>1</sup>https://github.com/BitVM/rust-bitcoin-scriptexec

<sup>&</sup>lt;sup>2</sup>https://github.com/Bitcoin-Wildlife-Sanctuary/bitcoin-circle-stark

File	Benchmark	LOC	Result	Time (s)	File	Benchmark	LOC	Result	Time (s)
	zip	36	~	2.03	(BSE)	bigint/sub	180	~	2.14
	copy	18	~	2.02		toaltstack	9	<b>v</b>	1.96
	roll	36	~	2.02		push modulus	9	~	1.97
	drop	5	~	2.00		fromaltstack	18	~	1.99
bigint/	is_zero_ke	37	~	2.16		div2	4,547	~	4.85
std	is_one_ke	38	~	2.21		div3	5.845	~	95.41
(BSE)	toaltstack	9	~	1.99		convert to be u4	4.007	~	28.13
	fromaltstack	18	~	2.01		convert to be bits	3.297	~	4.22
	is_negative	4	~	2.16		convert to be bits ta	3.081	~	4.28
	is_positive	4	X	2.16		convert to le bits	3,297	~	4.31
	resize	12	~	1.92		convert to le bits ta	3.535	~	4.39
	overall	18	100%	2.06		push zero	5	~	1.96
	add	173	<b>v</b>	3.58		push one not mtg	6	~	1.98
1	double allow overflow ke	134	~	3.58		sub	396	~	15.12
bigint/	double prevent overflow	133	~	3.54	bn254/	double	350	~	2.30
add	lshift prevent overflow	3,713	~	5.54	fp254impl	is zero ke	37	~	2.14
(BSE)	add ref with top	164	~	4.03	(BSE)	is one ke	52	~	2.13
	overall	863	100%	4.05		is one ke not mtg	95	~	2.12
	1 1.	0.007		4.00		is one not mtg	77	~	2.18
	convert_to_be_bits	3,297	V	4.38		inv	5,235,924	?	TO
bigint/	convert_to_te_bits	5,297	•	4.35		mul by constant	101.871	?	TO
bits	convert_to_be_bits_ta	3,081	V	4.54		square	133.523	?	TO
(BSE)	limb from botos	3,335	~	4.50		mul	136,960	?	TO
	init_irom_bytes	121	10007	2.14		mul bucket	71,852	?	TO
	overall	2,000	100%	5.91		decode_mtg	63,322	?	TO
	div2	4,157	~	6.26		convert to be bytes	67,363	?	TO
higint/	div2rem	4,156	~	6.48		mul by constant bucket	67,745	?	TO
inv	div3	5,057	~	2.29		overall	227,201	69%	10.08
(BSE)	div3rem	5,056	~	2.40		-hl- 0 1	0		0.11
(DOL)	inv_stage1	4,992,182	?	TO	6-14:	check_0_or_1	0	V	2.11
	overall	1,002,121	80%	4.36	(PCV)	decompose_positions_g	430	V	7.14
bigint/	equalverify	45	<ul> <li>✓</li> </ul>	2.13	(157)	skip_one_and_ext_bits_g	268	100%	7.14
cmp	lessthanoregual	183	~	2.13		overan	200	100%	7.07
(BSE)	overall	114	100%	2.13		limb_to_le_bits	376	~	2.48
		07		1.05		ltbbt_exc_low2b	351	~	2.13
	pusn_generator	2/		1.95		ltbbt_common	349	~	2.40
	pusn_zero	15	v	1.90	utils	qm31_reverse	3	~	2.00
bn254/	IS_ZEFO_KE	3/	2	2.1/ TO	(BSV)	ltbbt_exc_low1b	351	~	2.09
curves	auu	2,180,126	· ·	10 TO		dup_mv_g	64	~	2.05
(BSE)	double	947,166	2	10 TO		mv_from_bottom_g	96	~	2.05
	equaiverify	1,089,148	· · · · · · · · · · · · · · · · · · ·	10 TO		cta_top_item_first_in_g	28	~	2.00
	into_anine	5,/60,/11	1207	10		overall	202	100%	2.15
	overall	1,428,176	43%	2.03					
(BSE)	bigint/mul	102,932	~	452.10					

Table 3. Statistics and breakdown of bitguarb's performance for the full set of benchmarks. "TO" means timeout. For each small category, we show the averaged LOC, percentage of benchmarks solved and averaged time in the "overall" row.

benchmark. Only 13 (17%) 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).

Table 3 shows more details about the status of each benchmark and category. 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 3.91s. Even for programs in the bigint/inv category, which have an average of 1,002,121 LOCs, bitguard still managed to solve 80% of the benchmarks, with an average time of 4.36s. There are also some cases that are worth noting, for example, bigint/mul, which contains the most loops, but bitguard solves it within 452.10s, 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 13 benchmarks that **bitguard** fails to solve, we perform a manual analysis to identify the root causes. A vast majority of them (12 out of 13) 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 13 of them based on its internal strategy. Even after relaxing the



Fig. 7. A comparison between bitguard and its baselines without transpilation, where x-axis denotes the total number of benchmarks solverd, and y-axis denotes the cumulative time spent in seconds.

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 (65 out of 78, i.e., 83%) of benchmarks with a 12.55s 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 is no publicly available tool for verification of Bitcoin scripts, to evaluate the effectiveness of **bitguard**'s key design in Section 5.1, 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 presented in previous work [19]. That is, the baseline version doesn't perform any transpilation nor optimization. While it still shares the backend solver (Bitwuzla) with the default **bitguard**, we refer to this version as Baseline (Bitwuzla).

A subset of benchmarks (42.3%, 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 [29]. 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 [27] (i.e., cvc5–ff) as its backend solver. Specifically, for the 21 benchmarks that assume finite field inputs/outputs, **bitguarb** compiles them into finite field constraints and invokes cvc5–ff; for other benchmarks, cvc5 with default bitvector theory is used.

Figure 7 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 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 25.64% of benchmarks, thanks to the high-level DSL and the synthesis procedure discussed in Section 5.1. Baseline (cvc5/–ff) initially performs similarly to bitguard for the first 30 benchmarks but falls behind as more benchmarks are added, with bitguard ultimately solving 28.20% more benchmarks.



Fig. 8. An example code snippet demonstrating a bug in BitVM's implementation (a) and its fixed version (b).

**Result for RQ2:** bitguard performs significantly better than its ablative versions, with notable efficiency gains in 25.64% - 43.60% of cases. Thus, bitguard's design is important for its overall performance, and we believe that this answers RQ2 in a positive way.

## 7.3 Detecting Previously Unknown Vulnerabilities (RQ3)

Recall that Table 2 shows a benchmark (is\_positive) that could not be successfully verified. Further analysis revealed that the verification failure is due to a subtle, previously undocumented issue in the implementation.

As shown in Figure 8(a), the is\_positive function used OP\_LESSTHAN with the threshold HEAD\_OFFSET >> 1 to check if a bigint was "not negative". Here, HEAD\_OFFSET >> 1 serves as a midpoint: if the most significant limb of the bigint is less than this threshold, it indicates the sign bit is 0, meaning the number is non-negative. However, this approach mistakenly classified an all-zero number as positive because zero also has a most significant limb below HEAD\_OFFSET >> 1. To correct this, in Figure 8(b), the revised code adds an explicit zero check (Self::is\_zero\_keep\_element(depth)) and uses OP\_NOT to exclude zero values from being positive. The final check combines OP\_LESSTHAN with the inverted zero check using OP\_BOOLAND, ensuring that only non-zero, non-negative numbers are considered positive.

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. For instance, in the original code, zero could incorrectly pass the check, potentially allowing unintended validations where zero should have been excluded.

We promptly informed the developers of the affected libraries, and our finding was confirmed by the team. These results highlight the critical role of bitguard's verification design in identifying logical flaws that might otherwise lead to security vulnerabilities by allowing the acceptance of invalid proofs.

**Result for RQ3: bitguard** detected a previously unknown vulnerability in widely-used Bitcoin scripts, which could allow invalid proofs to be mistakenly accepted as valid.

# 8 Related work

*Formal methods for cryptography.* There is extensive research on applying formal verification techniques to cryptographic protocols. For example, Corin et al.[10] utilized a variant of probabilistic Hoare logic to verify the security of ElGamal, while Gagne et al.[15] applied similar methods to analyze the security of CBC-based MACs, PMAC, and HMAC. Tiwari et al. [34] 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  $\Sigma$ -protocols, which includes zk-SNARKs, using Isabelle/HOL [26] for formal correctness proofs. Sidorenco et al.[32] 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. [17] 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. [28] recently proposed a finite field solver that does not scale well in our benchmarks due to too many complex constraints. Finally, Coda [22] proposed the first verifier for the functional correctness of ZKP circuits. However, compared to **bitguarb**, 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 [11] 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 [37] 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)[24] has become an essential tool for symbolic reasoning, driven by the availability of practical, high-performance solvers like Z3[12], CVC4[3], and Gurobi[16]. The programming languages community has extensively explored the use of solvers for both verification and synthesis [20, 31, 33]. 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)[35, 36] automatically generate these constraints through symbolic compilation. One example is the Rosette framework[35], 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 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 78 benchmarks from BitVM implementations, our tool successfully verified 83% of the cases efficiently and identified one previously unknown vulnerability. 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

- 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] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct Non-Interactive zero knowledge for a von neumann architecture. In 23rd USENIX Security Symposium (USENIX Security 14), pages 781–796, San Diego, CA, Aug. 2014. USENIX Association.
- [7] Bitcoin. Bitcoin script. https://en.bitcoin.it/wiki/Script, 2022.
- [8] V. Buterin et al. A next-generation smart contract and decentralized application platform. white paper, 3(37):2-1, 2014.
- [9] 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.
- [10] 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.
- [11] F. Dahlgren. It pays to be circomspect. https://blog.trailofbits.com/2022/09/15/it-pays-to-be-circomspect/, 09 2022.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] L. Gurobi Optimization. Gurobi optimizer reference manual, 2019.
- [17] T. Hader. Non-linear smt-reasoning over finite fields, 2022.
- [18] C. A. R. Hoare. An axiomatic basis for computer programming. Commun. ACM, 12(10):576–580, 1969.
- [19] R. Klomp and A. Bracciali. On symbolic verification of bitcoin's script language. In J. Garcia-Alfaro, J. Herrera-Joancomartí, G. Livraga, and R. Rios, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 38–56, Cham, 2018. Springer International Publishing.
- [20] 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.
- [21] R. Linus, L. Aumayr, A. Zamyatin, A. Pelosi, Z. Avarikioti, and M. Maffei. BitVM2: Bridging bitcoin to second layers, Aug. 2024.
- [22] 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.
- [23] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Satoshi Nakamoto, 2008.
- [24] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. J. ACM, 27(2):356–364, Apr. 1980.
- [25] A. Niemetz and M. Preiner. Bitwuzla. In C. Enea and A. Lal, editors, Computer Aided Verification, pages 3–17, Cham, 2023. Springer Nature Switzerland.

- [26] T. Nipkow, M. Wenzel, and L. C. Paulson. Isabelle/hol: A proof assistant for higher-order logic. 2002.
- [27] A. Ozdemir. Cvc5-ff. https://github.com/alex-ozdemir/CVC4/tree/ff, 2022.
- [28] 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.
- [29] 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.
- [30] Polygon. Scalable payments, with zero-knowledge rollups. https://hermez.io, 2022.
- [31] 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.
- [32] 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.
- [33] A. Solar-Lezama. Program Synthesis by Sketching. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.
- [34] 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.
- [35] 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.
- [36] 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.
- [37] 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.