Programmable Bitcoin Verification via Synthesis-Aided Lifting

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

A new wave of proposals, such as covenants (OP_CHECKTEMPLATEVERIFY), the reactivation of OP_CAT, and BitVM-style fraud proofs, promises to turn Bitcoin into a settlement layer for decentralised finance. These projects must be expressed in Bitcoin script, a stack language with no loops or recursion; practical applications therefore expand into megabytes of unrolled opcodes whose slightest error can freeze or steal funds. Existing verification tools collapse under the resulting explosion of constraints.

We present 𝔟𝔦𝔱𝔤𝔲𝔞𝔯𝔡, the first scalable verifier for *programmable-Bitcoin* artifacts. Inspired by recent advances in program lifting and synthesis, 𝔟𝔦𝔱𝔤𝔲𝔞𝔯𝔡 automatically (i) lifts raw script into a semantics-preserving, register-based DSL and (ii) detects repetitive slices that mimic batch operations (map, fold, filter, Merkle-proof checks). Each slice is replaced with a single higher-order combinator whose behavior is captured by an axiom, shrinking downstream SMT constraints by orders of magnitude. A counter-example-guided inductive synthesis loop ensures every transformed fragment remains equivalent to its original script.

Evaluated on 104 real-world benchmarks, including full BitVM2 prover–verifier pairs, 𝔟𝔦𝔱𝔤𝔲𝔞𝔯𝔡 automatically verifies 88% of the cases in an average of 23.57s, outperforms direct SMT encodings by up to two orders of magnitude, and uncovers **5** previously unknown vulnerabilities. These results demonstrate that synthesis-aided lifting with axiomatized batch combinators delivers practical, rigorous assurance for the emerging ecosystem of programmable Bitcoin.

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

1 Introduction

Bitcoin [Nakamoto 2008] has earned its reputation as the most secure and decentralised ledger in existence, yet the ecosystem's aspirations have recently shifted from mere value transfer toward a fully fledged platform for programmable money. Proposals such as covenants (OP_CHECKTEMPLATEVERIFY) [Rubin 2020], the reactivation of OP_CAT [Heilman and Sabouri 2023], and the BitVM fraud-proof framework [Linus et al. 2024] all point to the same goal: execute sophisticated logic off-chain or under tight on-chain templates while letting Bitcoin's conservative

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; Yanju Chen, yanju@ucsb.edu, University of California, Santa Barbara; Yu Feng, yufeng@ucsb.edu, Nubit and University of California, Santa Barbara.

base layer arbitrate disputes. If successful, these ideas would unlock decentralized exchanges, lending markets, and roll-ups that settle directly in BTC, without abandoning Bitcoin's minimalist design philosophy. The impact is already tangible [Hiro Systems PBC 2025; IOV Labs 2025; Linus et al. 2024; Sovryn Community 2025]: the Rootstock side-chain today secures roughly $258 million in total value locked (TVL), Stacks exceeds $107 million, and Sovryn's non-custodial margin-trading system alone controls about $65 million. These nine-figure commitments underscore that programmable Bitcoin is no longer aspirational; instead, it is a production reality that demands strong correctness guarantees.

Yet developers face a steep *correctness cliff*. Unlike other platforms such as Ethereum [Buterin et al. 2014], which provides a Turing-complete programming model and built-in support for common cryptographic primitives such as elliptic curves and hash functions, Bitcoin script is deliberately non-Turing-complete, offers neither loops nor native cryptographic primitives, and exposes every operation through an untyped stack. Practical applications must therefore unroll every control-flow construct into megabytes of opcodes and then shard the result across taproot branches to respect Bitcoin's 4 MB block limit. For instance, implementing a standard zero-knowledge SNARK verifier [Ben-Sasson et al. 2014] that requires only 200 lines of Solidity code on Ethereum could result in a Bitcoin script program with several gigabytes. The stakes are painfully real: in June 2025 the Stacks-based ALEX protocol lost approximately $8.3 million (who had already suffered a $4.3 million bridge exploit the previous year [Cointelegraph 2024, 2025]); Rootstock's Sovryn lending pool was drained of roughly $1 million in October 2022 [Behnke 2022]; and the pNetwork cross-chain bridge leak of 277 BTC (about $13 million at the time) in September 2021 remains one of the largest Bitcoin-denominated DeFi breaches to date [Behnke 2021]. In an environment where a single misplaced opcode can freeze or steal funds, manual testing alone is manifestly inadequate.

Off-the-shelf verification pipelines struggle with these artifacts. Stack-oriented bytecode hides data flow, and naively encoding the resulting traces as Satisfiability Modulo Theories (SMT) constraints overwhelms modern solvers: verifying a 256-bit BitVM transcript directly in SMT can exhaust memory after hours (Section 7). Two empirical observations guide our solution. First, despite contorted stack manipulations, most script fragments implement register-style logic and can therefore be lifted to a cleaner intermediate form. Second and central to this paper, the apparent code explosion stems from repetitive yet simple patterns that mimic batch operations such as map, fold, and filter, idioms not expressible in raw Bitcoin script. Because each iteration performs the same straightforward arithmetic or boolean step, checking its equivalence to a concise summary is *far easier* than reasoning about an arbitrary low-level optimization; the difficulty lies in recognizing and compressing the repetition, not in proving the arithmetic itself.

Motivated by recent successes in program lifting and synthesis [Cheung et al. 2013], our solution assembles an end-to-end verification workflow that connects raw Bitcoin script to modern constraint solvers while keeping the original semantics intact. It unfolds in three stages. **First**, we introduce a register-style domain-specific language, $\mathcal{G}$, whose instruction set faithfully covers every opcode required by covenants and BitVM contracts, yet removes the clutter of explicit stack manipulation. **Second**, a two-tier lifting engine rewrites script into $\mathcal{G}$: a lightweight peephole pass translates local stack patterns, after which a counter-example-guided inductive-synthesis (CEGIS) loop [Solar-Lezama 2008] iteratively proposes, checks, and refines larger fragments until they match their script originals. **Third**, $\mathcal{G}$ provides a compact library of higher-order combinators (map, fold, zip, filter, etc.), each accompanied by a precise axiom that captures its observable behaviour. During lifting, the workflow detects repetitive "batch" slices in the script trace that correspond to these patterns and replaces thousands of unrolled instructions with a single axiomatized combinator call. Because the solver reasons directly over the axiom rather than the low-level code, the resulting SMT constraints shrink by orders of magnitude; equivalence checks collapse to short algebraic identities

that modern engines discharge in seconds. In practice, this strategy turns proofs that previously timed out, such as a 256-bit BitVM2 multiplication transcript, into sub-minute verification tasks (§7).

To evaluate our approach, we applied our tool to five real-world Bitcoin infrastructures, using a suite of 104 benchmarks derived from various implementations of BitVM and other cryptographic protocols. Our tool successfully verified 88% of the cases, demonstrating both its effectiveness and practicality. The verification process is efficient, with an average runtime of 23.57 seconds per benchmark.

Finally, bitguard identified **5** previously unknown vulnerabilities, and our ablation study also demonstrates the benefit of our synthesis approach, especially on complex benchmarks.

In summary, our contributions are as follows:

- We formulate the verification challenge for programmable Bitcoin contracts and present bitguard, the first automated framework that bridges raw Bitcoin script and modern formal-methods tooling.
- We design $\mathcal{G}$, a semantics-faithful yet analysis-friendly DSL equipped with higher-order constructs that succinctly summarise unrolled batch operations.
- We develop a CEGIS procedure that lifts large script artifacts to $\mathcal{G}$ and proves their equivalence, enabling scalable verification without manual annotations.
- We empirically demonstrate that bitguard scales to multi-megabyte transcripts, uncovers latent vulnerabilities, and offers a practical path toward safer programmable Bitcoin.

2 Background

To motivate our verification strategy, we first summarize the Bitcoin execution model, the constraints of its scripting language, the proposals that extend its programmability, and the security failures that make rigorous analysis essential.

Execution environment. Bitcoin maintains an *unspent-transaction-output* (UTXO) ledger: each transaction consumes one or more UTXOs and creates new ones. Spending a UTXO requires executing a two-part *script* program—an *unlocking* (witness) script supplied by the spender and a *locking* script stored in the UTXO. Both parts are concatenated and evaluated by every full node inside a sandbox whose resources are capped by consensus rules [Nakamoto 2008]: a maximum stack depth of 1 000 items, individual pushes limited to 520 bytes, and an aggregate 4 MB block size. Execution halts if the final stack top is non-zero; otherwise the transaction is invalid and the entire block is rejected.

Bitcoin script by example. To avoid DoS attacks, a Bitcoin script is intentionally tiny and non–Turing-complete. It has no loops or function calls, only about 110 opcodes, and all data flow passes through an untyped stack. The canonical pay-to-public-key-hash (P2PKH) output illustrates the style:

OP_DUP OP_HASH160 <pkh> OP_EQUALVERIFY OP_CHECKSIG

A spender supplies <sig> <pubkey> as the unlocking script; evaluation leaves true on the stack if (and only if) the signature is valid. More ambitious applications quickly hit the language limits. For example, a 256-bit modular multiply that occupies ≈200 lines of Solidity on Ethereum expands to more than 10 MB of inlined script because every loop iteration, conditional branch, and limb operation must be written out explicitly.

Extending expressiveness without a hard fork. Three proposals collectively known as the *programmable-Bitcoin tool kit* push expressiveness while preserving consensus:

(1) **Covenants** via OP_CHECKTEMPLATEVERIFY (BIP-119) [Rubin 2020], which fixes the shape of the future spending transaction.
(2) OP_CAT (draft BIP-420) [Heilman and Sabouri 2023], a cheap concatenation opcode that enables limited transaction introspection.
(3) **BitVM2** [Linus et al. 2024], which moves arbitrary computation off-chain and lets script enforce correctness through a logarithmic fraud-proof game.

These building blocks already underpin high-value Bitcoin-centric DeFi (*BTCFi*) systems: Stacks secures roughly $110M in total value locked (TVL), Rootstock $\sim \$250M$, and cross-chain bridges such as ALEX routinely clear eight-figure volumes.

What can go wrong. Incorrect script or bridge logic has led to substantial losses. The Stacks-based ALEX protocol suffered a $4.3 M bridge exploit in 2024 and a $8.3M validator bug in 2025 [Cointelegraph 2024, 2025]; Rootstock's Sovryn lending pool lost $1M in 2022 [Behnke 2022]; and the pNetwork bridge leak of 277 BTC ($13 M) in 2021 remains one of the largest Bitcoin-denominated breaches [Behnke 2021]. Because affected contracts are immutable, remediation required emergency treasuries, user bail-outs, or protocol-level rollbacks, none of which is ideal for a settlement layer trusted with billions of dollars.

Verification obstacles. Three technical hurdles combine to thwart conventional analyzers:

- **Implicit State** All variables live on the stack; recovering a data-flow graph requires untangling hundreds of OP_SWAP, OP_ROT, and arithmetic opcodes.
- **Code Explosion** Loop unrolling, table look-ups, and hand-rolled cryptographic kernels inflate contracts into multi-megabyte artifacts that must be split across Merkle branches to fit the block limit.
- **Solver Blowup** A direct SMT encoding of a 256-bit BitVM2 verifier yields millions of non-linear constraints and typically times out or runs out of memory in state-of-the-art solvers such as cvc5 [Barbosa et al. 2022; Ozdemir 2022] or Bitwuzla [Niemetz and Preiner 2023].

Nonetheless, the underlying computations are often repetitive and straight-line. For instance, a sequence of identical limb additions that collectively implements a map or fold. Capturing these patterns at a higher level and verifying them as wholes, rather than as billions of individual stack mutations, is the central idea pursued in the rest of this paper.

3 Overview

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

Motivating example. Big integer (BigInt) multiplication is a fundamental operation underpinning critical cryptographic primitives, including RSA encryption and signatures, elliptic curve cryptography, zero-knowledge proofs, and homomorphic encryption. Figure 1(a) illustrates a BitVM implementation of BigInt multiplication written in Bitcoin script, whose functional correctness we aim to formally verify. Specifically, given two BigInts A and B, the correctness verification problem can be succinctly expressed by the following Hoare triple:

$$\mathsf{BigInt}(A) \wedge \mathsf{BigInt}(B) \quad \{C(A,B)\} \quad R = C(A,B) \wedge R = A \cdot B, \tag{1}$$

where C corresponds to the program being verified, R is the output, and BigInt type checks the given number as big integer.

As with many multi-precision arithmetic techniques used for representing very large integers, the big integer implementation in this example divides numbers into smaller slices (or formally:

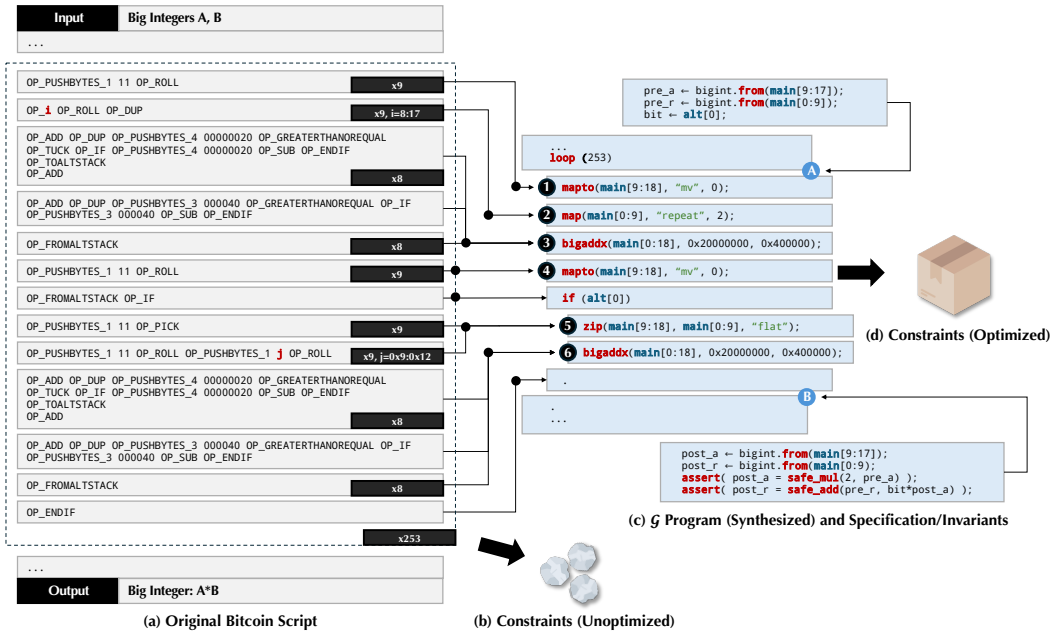


Fig. 1. A motivating example showing a partial Bitcoin script for computing big integer multiplication and its corresponding $\mathcal{G}$ program synthesized by bitguard. One then checks the correctness of the snippet by providing specification written in $\mathcal{G}$, which produces optimized constraints.

limbs) to accommodate the limited word size of the underlying virtual machine. Specifically, a BigInt A is represented as a 254-bit integer divided into 9 slices $(A[0], \ldots, A[8])$, where each slice contains 29 bits, except the final slice, which has 22 bits. Formally, the integer A can be expressed as:

$$A = \sum_{i=0}^{8} 2^{29 \cdot i} \cdot A[i].$$

Therefore, multiplication of two BigInts A and B then corresponds to summary of each slice:

$$A \cdot B = (A[0] \cdot B[0] \pmod{2^{29}}, \quad A[1] \cdot B[1] + c_0 \pmod{2^{58}}, \quad \ldots),$$

where c_i corresponds to the carry of the multiplication of the i-th slices.

Challenges. However, as shown by Figure 1, the multiplication is further broken down into low-level stack operations that are non-straightforward due to the limitation of the Bitcoin script's stack-based virtual machine. In order to store the result on top of the stack, one has to break B up into its binary representation stored in an alternative stack, and invokes a loop of 253 iterations that multiplies each bit from B with slices of A step by step. This entangles the high-level BigInt multiplication semantics and produces a large number of non-linear constraints during verification, making it difficult to scale.

Key observations and insights. Figure 2 illustrates the changes of stacks during the execution of a single loop iteration in BigInt multiplication, where we can see a series of high-level semantic patterns if we mark out BigInt A, B and the multiplication result R. Here, Bitcoin Script's stack-based execution uses both a *main stack* and an *alt stack*, where the alt stack serves as auxiliary storage for temporarily saving and restoring intermediate values, to manage all data through explicit stack

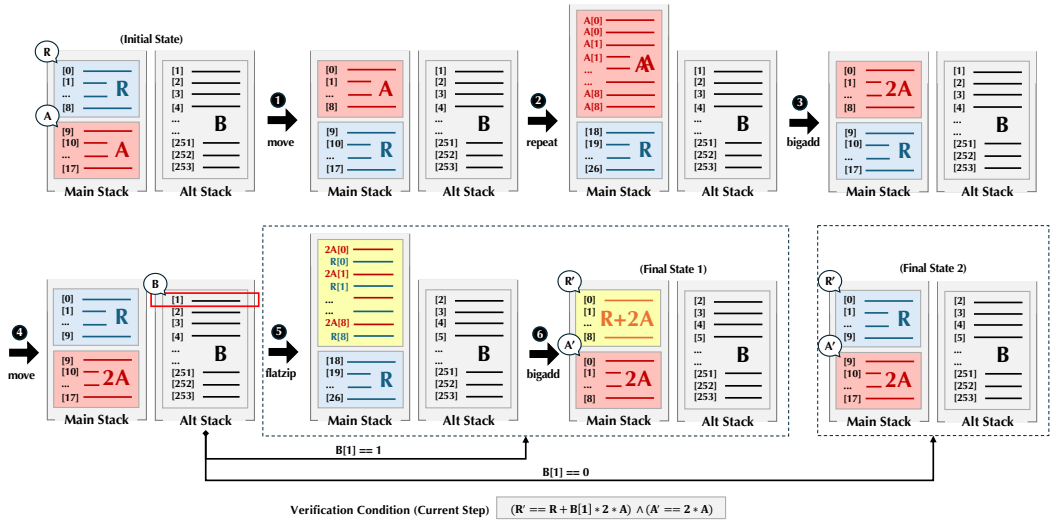


Fig. 2. Illustration of the operations within a single loop iteration of big integer multiplication.

operations. In order to compute such a step of $A \cdot B[0]$ and put the result into R, one needs to move A to the top of stack and repeat itself, before multiplication with $B[0]$ and setting it to the result. Such a sequence actually corresponds to the loop body as shown in Figure 1 but comes in such a way that its semantics can be concisely formalized to reduce the complexity for verification; it soundly provides the semantics with higher-order functions without analysis of low-level computation.

Our solution: lifting to a high-level DSL. Based on the above observation, we propose lifting the original low-level Bitcoin script to a high-level domain-specific language (DSL). Inspired by recent successes in program synthesis [Cheung et al. 2013; Solar-Lezama 2008], our key insight is to synthesize and lift Bitcoin script into its equivalent high-level representation. Figure 1(b) shows the equivalent version of the BigInt multiplication script in our DSL $\mathcal{G}$. Note that this approach abstracts away the complexity of stack manipulation by converting the original script into a more concise and understandable three-address code format with a clean loop structure that is easy to verify.

Verification for functional properties. With the synthesized $\mathcal{G}$ program, we can verify its correctness. Specifically, given a Hoare triple $\{P\}C\{Q\}$ as shown in Equation 1, where P is the precondition, Q is the postcondition, C is the program in our high-level DSL, we reduce the non-linear constraints generated by the original script into simpler, tractable verification conditions as follows:

- **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: $R' = A \cdot B \wedge A' = 2^{253} \cdot A[0]$, where R' and A' denote the resulting states after all iterations.
- **Loop Invariant** After lifting the code to our high-level DSL, we leverage the Houdini algorithm [Flanagan and Leino 2001] to synthesize the loop invariant—a logical condition

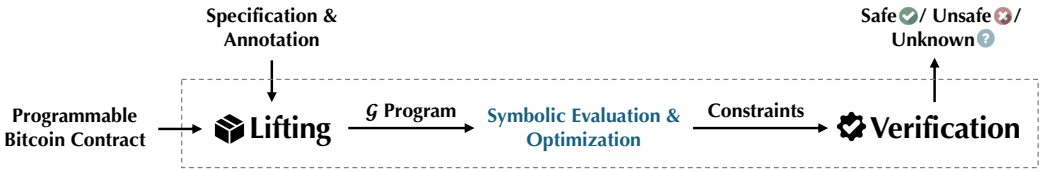


Fig. 3. A high-level overview of the bitguard verification framework.

that holds true before and after each iteration of a loop. The loop invariant for this multiplication ensures that after each iteration: $R' = R + B[i] \cdot 2A \wedge A' = 2A$. 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:

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

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 by an off-the-shelf SMT solver [Barbosa et al. 2022; de Moura and Bjørner 2008], 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 bitguard Framework

In this section, we introduce the overall verification algorithm of bitguard. We first describe a high-level overview of the system (Section 4.1), including its key procedures. Then we introduce the domain-specific language $\mathcal{G}$ built within bitguard (Section 4.2) and its symbolic evaluation rules (Section 4.3), which can be used to summarize stack-based operations in a verification-friendly way. As an improvement to verification, $\mathcal{G}$ 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 [Linus et al. 2024] and user-provided specifications. It then outputs whether the given system is safe regarding the specification, in particular, in three potential outcomes: safe (✔), unsafe (✘) or unknown (?). Specifically, bitguard contains two major phases:

- **Lifting** Bitcoin scripts are notoriously difficult to analyze directly due to their stack-based, low-level nature. To overcome this challenge, bitguard lifts the original script into a high-level, register-style intermediate representation with the proposed $\mathcal{G}$ language using program synthesis. This lifting process automatically recognizes local stack patterns and iteratively rewrites larger fragments into concise $\mathcal{G}$ code, while detecting repetitive structures that correspond to batch operations such as map, fold, and filter. These patterns are then replaced by succinct combinators in $\mathcal{G}$, significantly reducing downstream verification

s	::=		**Statement:**	μ	$\in$	$\{\text{main}, \text{alt}, \text{ext}\}$	**Context Selectors**
	|	$s*$	sequence	σ	::=		**Stack Operation:**
	|	e;	expression		|	$\text{append}(c*)$	stack append
	|	σ;	stack operation		|	$\text{switch}(k)$	move bt. stacks
	|	if (e) then s else s.	branch		|	$\text{mapto}(k, \bullet, c)$	stack mapto
	|	loop (c) s.	loop		|	$\text{filter}(k, \otimes, c)$	stack filter
	|	$i \leftarrow e$;	assignment		|	$\text{map}(k, \bullet, c)$	stack map
	|	$\text{assume}(e)$;	assumption		|	$\text{fold}(k, \bullet, c)$	stack fold
	|	$\text{assert}(e)$;	assertion		|	$\text{zip}(k, k, \bullet)$	stack zip
e	::=		**Expression:**	$\bullet$	$\in$	$\{\otimes, \oplus, \odot, \oslash, \ominus\}$	**Operators**
	|	i	identifier	$\otimes$	$\in$	$\{\wedge, \vee, =, \neq, <, \leq, \ldots\}$	**Boolean Ops.**
	|	c	constant	$\oplus$	$\in$	$\{+, -, *, /, \ldots\}$	**Integer Ops.**
	|	$\diamond$	symbolic	$\odot$	$\in$	$\{\text{sha1}, \text{hash160}, \ldots\}$	**Hashing Ops.**
	|	$\mu[c*]$	context accessor	$\oslash$	$\in$	$\{\text{cat}, \text{size}, \ldots\}$	**String Ops.**
	|	$e \bullet e \mid \bullet e \mid \bullet$	operations	$\ominus$	$\in$	$\{\text{mv}, \text{cp}, \text{flatzip}, \ldots\}$	**Stack Ops.**

Fig. 4. A representative set of the syntax of $\mathcal{G}$ programs. For clarify, we omit the OP_ prefix in Bitcoin script operators, e.g., cat for OP_CAT.

complexity. The result is a $\mathcal{G}$ program that faithfully captures the original script's semantics while enabling scalable formal reasoning.

- **Verification** The user provides correctness specifications (such as preconditions, post-conditions, and verification conditions) directly within the synthesized $\mathcal{G}$ program using $\mathcal{G}$'s dedicated verification interface (e.g., assume and assert). To handle loops appearing in the original Bitcoin script, `bitguard` lifts them into corresponding higher-order functional constructs. In the subsequent *verification phase*, `bitguard` leverages symbolic evaluation rules to transform the annotated $\mathcal{G}$ program into logical constraints, which can then be efficiently analyzed by an off-the-shelf constraint solver.

We first elaborate on the verification procedure and defer a detailed discussion of the lifting phase to Section 5. Specifically, we give an introduction of the $\mathcal{G}$ language in Section 4.2, including its core syntax for modeling program behavior and writing verification queries. Building on top of $\mathcal{G}$, we then describe how an $\mathcal{G}$ 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 $\mathcal{G}$ program into logical constraints, thus reducing a verification task into constraint solving.

4.2 The $\mathcal{G}$ Language for Modeling Stack Operations

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

- **Specification as Correctness Properties** $\mathcal{G}$ 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 e evaluates to true. In a $\mathcal{G}$ program, a verification query e can be built from $\mathcal{G}$ expressions, and tracked with the assignment construct $i \leftarrow e$ in a dedicated environment besides the stacks.

Example 4.1 (Properties). For the BigInt multiplication example shown in Figure 1, the property from Equation 1 can be expressed in the following way:

$$\mathsf{assert}(A' = \mathsf{safe\_mul}(2, A[0]));$$

$$\mathsf{assert}(R' = \mathsf{safe\_add}(R, B[0] * A[0]));$$

where safe_mul and safe_add are safe arithmetic multiplication and addition operations that prevent overflows and underflows.

- **Basic Types and Control Flows** There are three basic types in $\mathcal{G}$, namely booleans, integers and hashes. In addition to standard arithmetic operators for booleans ($\otimes$) and integers ($\oplus$), $\mathcal{G}$ also models cryptographic operations ($\odot$) such as sha1 and hash160, which compute hashes as their output, as well as string operations ($\oslash$) that transform and obtain properties of strings. $\mathcal{G}$ models standard control flows such as branches and loops. Note that a loop in $\mathcal{G}$ by default has a constant bound c (i.e., *bounded*) due to the nature of stack-based scripts.
- **Stack Operations** $\mathcal{G}$ 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 $\bullet$ 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 $\bullet$.
  - The **zip** operator is a higher-order operator, which takes two subsets of stack elements and applies a function $\bullet$ to each pair of them. The resulting set of elements is then pushed to the top of the stack.

Example 4.2 (A Program in $\mathcal{G}$). The following shows a $\mathcal{G}$ program:

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

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 [Flanagan and Leino 2001] 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 conjunctive invariants over this universe in the standard way.

4.3 Symbolic Evaluation for the $\mathcal{G}$ Language

We build upon the existing symbolic operational semantics for Bitcoin script defined by Klomp and Bracciali [2018]. Specifically, we adopt their formalization to capture the symbolic side effects

produced during script execution and integrate these effects into our defined *program state*. Formally, we represent the resulting state as a 4-tuple $\langle p, \gamma, \delta, \pi \rangle$, where:

- p is the *program counter* that points to the immediate next statement.
- γ is the *assertion store* that tracks verification conditions generated during symbolic evaluation, which can be implied by the language constructs or derived from user-provided specification.
- δ is the *program store* that provides access to the memory and stacks. Specially, a stack operation σ can access both the main and alt stacks by the form $\delta[\mathsf{main}]$ and $\delta[\mathsf{alt}]$; some operators like OP_CHECKTEMPLATEVERIFY can access information from the external context $\delta[\mathsf{ext}]$, such as the transaction that the current script is in (via $\delta[\mathsf{ext}].\mathsf{tx}$), the message sender (via $\delta[\mathsf{ext}].\mathsf{sender}$), etc.; besides, the verification interface can access the memory with given identifier i, in the form $\delta[i]$.
- π 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 π, we then say x is *guarded* by π, denoted by $(\!|\pi|\!)x$. Thus, each slot i of the program store δ, also denoted as $\delta[i]$, is mapped to a set of possible values guarded by different path conditions:

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

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

$$\delta[\![i]\!] = \delta_\pi[\![i]\!] = \{(\!|\pi'|\!)x \in \delta[i] \mid \pi' \Rightarrow \pi\}.$$

We then describe how bitguard *symbolically* evaluates a $\mathcal{G}$ program and keeps track of program states via a set of evaluation rules. The $\mathcal{G}$ language shares the same formulation of program state of the Bitcoin script language.

Control flows and verification interface. Figure 5 shows a representative set of symbolic evaluation rules for the control flow constructs and verification interface of $\mathcal{G}$. 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 δ. 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 π 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

$$\frac{\langle s_0, \gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \gamma_0, \delta_0, \pi_0\rangle \quad \ldots \quad \langle s_n, \gamma_{n-1}, \delta_{n-1}, \pi_{n-1}\rangle \rightsquigarrow \langle \varnothing, \gamma_n, \delta_n, \pi_n\rangle}{\langle (s_0, \ldots, s_n), \gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \gamma_n, \delta_n, \pi_n\rangle}\ (\textsc{Seqn})$$

$$\frac{}{\langle c, \gamma, \delta, \pi\rangle \rightsquigarrow \langle c, \gamma, \delta, \pi\rangle}\ (\textsc{Cnst}) \qquad \frac{}{\langle \diamond, \gamma, \delta, \pi\rangle \rightsquigarrow \langle \diamond, \gamma, \delta, \pi\rangle}\ (\textsc{Symb}) \qquad \frac{}{\langle i, \gamma, \delta, \pi\rangle \rightsquigarrow \langle \delta[\![i]\!], \gamma, \delta, \pi\rangle}\ (\textsc{Iden})$$

$$\frac{\begin{array}{c}\langle e, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v, \gamma_e, \delta_e, \pi_e\rangle \\ \langle p_0, \gamma_e, \delta_e, \pi_e \wedge v\rangle \rightsquigarrow \langle \varnothing, \gamma_0, \delta_0, \pi_0\rangle \\ \langle p_1, \gamma_e, \delta_e, \pi_e \wedge \neg v\rangle \rightsquigarrow \langle \varnothing, \gamma_1, \delta_1, \pi_1\rangle \\ \gamma' = \gamma_e \cup \gamma_0 \cup \gamma_1 \qquad \delta' = \delta_e \uplus \delta_0 \uplus \delta_1\end{array}}{\langle \text{if } (e) \text{ then } p_0 \text{ else } p_1, \gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi_e\rangle}\ (\textsc{Bnch})$$

$$\frac{\begin{array}{c}\langle e_0, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v_0, \gamma_0, \delta_0, \pi_0\rangle \\ \langle e_1, \gamma_0, \delta_0, \pi_0\rangle \rightsquigarrow \langle v_1, \gamma', \delta', \pi'\rangle \\ v_0, v_1 \in \text{booleans} \cup \text{integers} \cup \text{strings} \\ \circ \in \{\otimes, \oplus\} \qquad v = v_0 \circ v_1\end{array}}{\langle e_0 \circ e_1, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v, \gamma', \delta', \pi'\rangle}\ (\textsc{Bexp})$$

$$\frac{\begin{array}{c}\langle e, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v, \gamma', \delta', \pi'\rangle \\ v \in \text{booleans} \cup \text{integers} \cup \text{strings} \cup \text{hashes} \\ \circ \in \{\odot, -, \neg\} \qquad v' = \circ v\end{array}}{\langle \circ e, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v', \gamma', \delta', \pi'\rangle}\ (\textsc{Uexp})$$

$$\frac{\langle e, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v, \gamma', \delta_0, \pi'\rangle \qquad \delta' = \delta_0 \uplus \{i \mapsto \{(\!|\pi|\!)v\}\}}{\langle i \leftarrow e, \gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi'\rangle}\ (\textsc{Asgn})$$

$$\frac{\langle e, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v, \gamma', \delta', \pi_0\rangle \qquad \pi' = \pi_0 \wedge v}{\langle \text{assume}(e), \gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi'\rangle}\ (\textsc{Asum})$$

$$\frac{\langle e, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v, \gamma_0, \delta', \pi'\rangle \qquad \gamma' = \gamma_0 \cup \{\pi' \Rightarrow v\}}{\langle \text{assert}(e), \gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi'\rangle}\ (\textsc{Asrt})$$

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

per each value mapping. Given program stores δ_0 and δ_1, their merged version $\delta_0 \uplus \delta_1$ is given by:

$$\begin{aligned}&\delta_0 \uplus \delta_1 = A \cup B \cup C,\\ &\text{where } A = \{i \mapsto \delta_0[i] \mid i \in \text{dom}(\delta_0) \backslash \text{dom}(\delta_1)\}, B = \{i \mapsto \delta_1[i] \mid i \in \text{dom}(\delta_1) \backslash \text{dom}(\delta_0)\},\\ &\quad \text{and } C = \{i \mapsto \delta_0[i] \cup \delta_1[i] \mid i \in \text{dom}(\delta_0) \cap \text{dom}(\delta_1)\}.\end{aligned}$$

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

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 δ to an identifier i. The assumption rule (Asum) adds the resulting value v of evaluation of the expression e into the current path condition π by conjunction. Similarly, the assertion rule (Asrt) appends v to the assertion store γ. During the evaluation, `bitguard` terminates when the current path condition π evaluates to false, or the conjunction of all clauses from the assertion store γ can not be satisfied.

Bitcoin script operators. Figure 6 formalizes the symbolic semantics of several key Bitcoin script operators used in programmable contracts. (OpCat) enables symbolic concatenation of two stack elements, supporting advanced protocols that require dynamic commitments. (OpDepth) provides stack introspection by pushing the current stack depth, which is critical for scripts that manipulate stack indices. (OpCheckTemplateVerify) enforces template-based spending via an uninterpreted predicate over the current transaction, enabling covenant-style restrictions without modeling full transaction structure. Finally, (OpSwap), (OpRot), (OpPick), and (OpRoll) are stack manipulation primitives that enable efficient batch processing and complex arithmetic by directly updating the stack layout. Together, these rules extend the verifier's ability to reason about modern programmable Bitcoin scripts involving advanced data manipulation, introspection, and template constraints.

$$\frac{\langle e_0, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v_0, \gamma_0, \delta_0, \pi_0\rangle \quad \langle e_1, \gamma_0, \delta_0, \pi_0\rangle \rightsquigarrow \langle v_1, \gamma', \delta', \pi'\rangle \quad v_0, v_1 \in \text{strings} \quad \circ \equiv \text{OP\_CAT} \quad v = v_0 \cdot v_1}{\langle e_0 \circ e_1, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v, \gamma', \delta', \pi'\rangle} \ (\textsc{OpCat}) \qquad \frac{\circ \equiv \text{OP\_DEPTH} \quad l = \delta[\![\text{main}]\!] \quad \delta' = \delta \uplus \{\text{main} \mapsto [|l|] \cdot l\}}{\langle \circ, \gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \gamma, \delta', \pi\rangle} \ (\textsc{OpDepth})$$

$$\frac{\langle e, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v_e, \gamma', \delta', \pi'\rangle \quad v_e \in \text{hashes} \quad \circ \equiv \text{OP\_CHECKTEMPLATEVERIFY} \quad v = \text{UF}(\delta[\text{ext}].\text{tx}, v_e)}{\langle \circ e, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v, \gamma', \delta', \pi'\rangle} \ (\textsc{OpCheckTemplateVerify})$$

$$\frac{\delta[\![\text{main}]\!] \equiv [v_0, v_1] \cdot l \quad \circ \equiv \text{OP\_SWAP} \quad \delta' = \delta \uplus \{\text{main} \mapsto [v_1, v_0] \cdot l\}}{\langle \circ, \gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \gamma, \delta', \pi\rangle} \ (\textsc{OpSwap}) \qquad \frac{\delta[\![\text{main}]\!] \equiv [v_0, v_1, v_2] \cdot l \quad \circ \equiv \text{OP\_ROT} \quad \delta' = \delta \uplus \{\text{main} \mapsto [v_2, v_0, v_1] \cdot l\}}{\langle \circ, \gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \gamma, \delta', \pi\rangle} \ (\textsc{OpRot})$$

$$\frac{\langle e, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v_e, \gamma', \delta_0, \pi'\rangle \quad v_e \in \text{integers} \quad \circ \equiv \text{OP\_PICK} \quad \delta_0[\![\text{main}]\!] = [v_0, ..., v_n] \cdot l \quad \delta' = \delta_0 \uplus \{\text{main} \mapsto [v_n, v_0, ..., v_n] \cdot l\}}{\langle \circ e, \gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi'\rangle} \ (\textsc{OpPick}) \qquad \frac{\langle e, \gamma, \delta, \pi\rangle \rightsquigarrow \langle v_e, \gamma', \delta_0, \pi'\rangle \quad v_e \in \text{integers} \quad \circ \equiv \text{OP\_ROLL} \quad \delta_0[\![\text{main}]\!] = [v_0, ..., v_n] \cdot l \quad \delta' = \delta_0 \uplus \{\text{main} \mapsto [v_n, v_0, ..., v_{n-1}] \cdot l\}}{\langle \circ e, \gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi'\rangle} \ (\textsc{OpRoll})$$

Fig. 6. A representative set of the symbolic evaluation rules (part 2) for the programmable Bitcoin script operators. Here, we define the concatenation of two elements v_1 and v_2 as $v_1 \cdot v_2$, and an uninterpreted function UF.

Batched stack operators. The symbolic evaluation rules of the higher-order constructs for modeling batched stack operations of $\mathcal{G}$ are shown in Figure 7. 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 the remaining four higher-order operations, namely the rules of (Mapto), (Filter), (Map), (Fold) and (Zip), accept an operator $\bullet$ that is used to transform the input elements into new ones.

Example 4.3 (Batch Processing with a $\mathcal{G}$ Construct: map*).* Given the following $\mathcal{G}$ program:

$$\text{map}(\text{main}[0{:}2], +, 1);$$

Suppose main[0:2] is $(1, 2)$ initially, execution of the program results in $(2, 3)$. For the symbolic version of the program:

$$\text{map}(\text{main}[0{:}2], +, \{(\!|\pi_0|\!)x, (\!|\pi_1|\!)y\});$$

with a stack that also contains symbolic values, e.g.,

$$\text{main}[0{:}2] = (\{(\!|\pi_1|\!)1, (\!|\pi_2|\!)2\}, 3),$$

execution of the $\mathcal{G}$ program yields a symbolic result main′[0:2] with the path conditions merged:

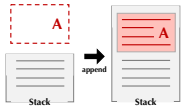
$$\text{main}'[0{:}2] = (\{(\!|\pi_0 \wedge \pi_1|\!)x + 1, (\!|\pi_0 \wedge \pi_2|\!)x + 2, (\!|\pi_1|\!)y + 1, (\!|\pi_1 \wedge \pi_2|\!)y + 2\}, \{(\!|\pi_0|\!)x + 3, (\!|\pi_1|\!)y + 3\}).$$

We then generalize the merging of path conditions in the above example during program execution as a general path merging rule: the result of applying an operator $\bullet$ on the guarded values g_0 and g_1 is given by:

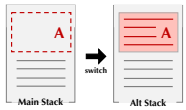
$$g_0 \bullet g_1 = (\!|\pi_0 \wedge \pi_1|\!)(x_0 \bullet x_1),$$

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

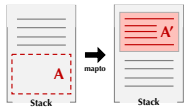
$$\frac{l = (c_0, ..., c_n) \qquad X = l \cdot \delta[\![\text{main}]\!] \qquad \delta' = \delta \cup \{\text{main} \mapsto X\}}{\langle \text{append}(l), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma, \delta', \pi \rangle} \ (\textsc{Append})$$



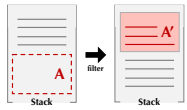
$$\frac{\begin{array}{c}\mu' = \text{alt if } \mu \equiv \text{main else main} \qquad M = \delta[\![\mu]\!] \qquad M' = \delta[\![\mu']\!] \\ A = \delta[\![\mu]\!][\![l]\!] \qquad X_0 = M \backslash A \qquad X_1 = A \cdot M' \\ \delta' = \delta \cup \{\mu \mapsto X_0, \mu' \mapsto X_1\} \qquad \gamma' = \gamma \cup \gamma_0\end{array}}{\langle \text{switch}(\mu[l]), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle} \ (\textsc{Switch})$$



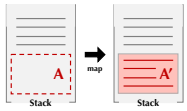
$$\frac{\begin{array}{c}M = \delta[\![\mu]\!] \qquad A = \delta[\![\mu]\!][\![l]\!] \qquad A' = \text{flatten}(\{v \bullet c \mid v \in A\}) \\ X = A' \cdot (M \backslash A) \qquad \delta' = \delta \cup \{\mu \mapsto X\} \qquad \gamma' = \gamma \cup \gamma_a\end{array}}{\langle \text{mapto}(\mu[l], \bullet, c), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle} \ (\textsc{Mapto})$$



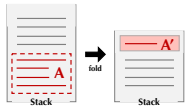
$$\frac{\begin{array}{c}M = \delta[\![\mu]\!] \qquad A = \delta[\![\mu]\!][\![l]\!] \qquad A' = \{v \mid v \otimes c = \text{true} \wedge v \in A\} \\ X = A' \cdot (M \backslash A) \qquad \delta' = \delta \cup \{\mu \mapsto X\} \qquad \gamma' = \gamma \cup \gamma_0\end{array}}{\langle \text{filter}(\mu[l], \otimes, c), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle} \ (\textsc{Filter})$$



$$\frac{\begin{array}{c}M = \delta[\![\mu]\!] \qquad X = \{M[\![i]\!] \bullet c \text{ if } i \in l \text{ else } M[\![i]\!] \mid 0 \le i \le |M|\} \\ \delta' = \delta \cup \{\mu \mapsto X\} \qquad \gamma' = \gamma \cup \gamma_0\end{array}}{\langle \text{map}(\mu[l], \bullet, c), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle} \ (\textsc{Map})$$



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



$$\frac{\begin{array}{c}l_a = (c_0, ..., c_n) \qquad l_b = (d_0, ..., d_n) \qquad M = \delta[\![\mu]\!] \qquad A = \delta[\![\mu]\!][\![l_a]\!] \\ B = \delta[\![\mu]\!][\![l_b]\!] \qquad X = \{A[\![c_i]\!] \bullet B[\![d_i]\!] \mid 0 \le i \le n\} \cdot (M \backslash A \backslash B) \\ \delta' = \delta \cup \{\mu \mapsto X\} \qquad \gamma' = \gamma \cup \gamma_b\end{array}}{\langle \text{zipwith}(\mu[l_a], \mu[l_b], \bullet), \gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \gamma', \delta', \pi \rangle} \ (\textsc{Zip})$$

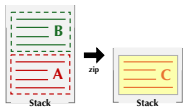


Fig. 7. A representative set of the symbolic evaluation rules (part 3) for the stack operations of $\mathcal{G}$. We illustrate changes to stacks before and after the corresponding stack operation to the right of each rule, and define the concatenation of two lists l_1 and l_2 as $l_1 \cdot l_2$, as well as the removal of list l_2 from top of l_1 as $l_1 \backslash l_2$.

5 Synthesis-Aided Lifting

In this section, we introduce the lifting algorithm that converts a Bitcoin script into its equivalent $\mathcal{G}$ program via a counterexample-guided inductive synthesis (CEGIS) loop. The synthesized $\mathcal{G}$ 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 (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 $\mathcal{G}$ and a Bitcoin script P, `bitguard` starts by obtaining the synthesis specification $\mathcal{S}_\phi$ via symbolic evaluation (i.e., the SEval procedure)

Algorithm 1 Synthesis-Aided Lifting

1: **procedure** LIFT($\mathcal{G}, P$)
2: **input:** domain-specific language $\mathcal{G}$, Bitcoin script P
3: **output:** lifted $\mathcal{G}$ program P' or $\perp$ if not found
4: $\mathcal{S}_\phi \leftarrow \text{SEVAL}(P)$ ▷ symbolically evaluates original script into logical specification
5: **sample** $E \sim \{(e_{\text{in}}, e_{\text{out}}) \mid P(e_{\text{in}}) = e_{\text{out}}\}$ ▷ samples input-output examples from script P
6: $\kappa \leftarrow \top$ ▷ initializes knowledge base
7: **while** $P' \leftarrow \text{ENUMERATE}(\mathcal{G}, E, \kappa)$ **do**
8: $\mathcal{S}' \leftarrow \text{SEVAL}(P')$ ▷ symbolically evaluates candidate program into logical constraints
9: $r \leftarrow \text{sat}(\mathcal{S}' \not\models \mathcal{S}_\phi)$ ▷ check for counterexample
10: **if** r **then**
11: $(e'_{\text{in}}, e'_{\text{out}}) \leftarrow \text{cex}(r), E \leftarrow E \cup (e'_{\text{in}}, e'_{\text{out}})$ ▷ gets the counterexample and adds to example set
12: $\kappa \leftarrow \kappa \wedge \text{block}(P')$ ▷ blocks the current candidate program
13: **else return** P' ▷ no counterexample is found; returns the program
14: **return** $\perp$ ▷ exhausted

of P (line 4). It then samples an initial set E of input-output examples from the original script P (line 5). Each example $(e_{\text{in}}, e_{\text{out}})$ consists of an input e_{in} and an output e_{out} that correspond to the status of the stacks before and after applying the script P respectively, i.e., $P(e_{\text{in}}) = e_{\text{out}}$. bitguard then continuously constructs candidate $\mathcal{G}$ programs via the ENUMERATE procedure (line 7-14) until a solution is found. Specifically for each proposed candidate program P', bitguard obtains its representation $\mathcal{S}'$ in constraints (line 8) and checks if there exists an input-output example (i.e., a counterexample) from $\mathcal{S}'$ that violates the synthesis specification $\mathcal{S}_\phi$ (line 9). The candidate program P' is not the solution if such a counterexample $(e'_{\text{in}}, e'_{\text{out}})$ 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 script P (line 13).

5.2 The Enumeration Procedure

Given a domain-specific language, which here refers to $\mathcal{G} = (V, \Sigma, R, S)$, where V, Σ, R and S denote the non-terminals, terminals, productions and start symbol respectively, the enumeration procedure finds a *feasible* program P in $\mathcal{G}$, such that for all given input-output examples $(e_{\text{in}}, e_{\text{out}}) \in E$, 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 $\mathcal{G}$ 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 $\mathcal{G}$ by construction, we model P as a sequence of terminals V and non-terminals Σ in $\mathcal{G}$: $P \in (V \cup \Sigma)*$, such that P can be derived from S via a sequence of productions from R:

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

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 $\mathcal{G}$:

$$\text{mapto}(k_0, \text{“mv”}, 0);\ \text{zip}(k_1, k_2, \text{“flat”});$$

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

Table 1. A representative set of logical summary of $\mathcal{G}$. x denotes the input. m and a denote the main and alt stack respectively. σ_p denotes the size of p, and ϕ_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).

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:

$$\text{mapto}(\text{main}[9], \text{“mv”}, 0);\ \text{zip}(k_1, k_2, \text{“flat”});$$

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. Insipired methodology of component-based synthesis with *over-approximate* logical specifications [Feng et al. 2017], 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 $\mathcal{G}$, 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 $\Psi(P(e_{\text{in}}) = e_{\text{out}})$:

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

where ϕ_n denotes the logical summary of the node n.

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

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

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:

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

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 $(e_{\text{in}}, e_{\text{out}}) \in E$, if its logical encoding $\Psi(P(e_{\text{in}}) = e_{\text{out}})$ is unsatisfiable, then P can be safely pruned. Therefore, the enumeration procedure returns the program P, if the following query yields true:

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

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 script P. However, verifying this equivalence is non-trivial, as there is no off-the-shelf equivalence checker for comparing Bitcoin script with programs in $\mathcal{G}$. 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 [Klomp and Bracciali 2018] with those already defined in Section 4 for $\mathcal{G}$. The checker was built on top of the Rosette framework [Torlak and Bodik 2014] and leverages its SMT encoding facilities as well as its symbolic evaluation engine.

Given the described equivalence checking mechanism, we are now ready to formally establish the soundness of the proposed lifting-based verification method.

Theorem 5.3 (Soundness of Verification via Synthesis-Aided Lifting). Let P_{BTC} be a Bitcoin script program, and let $P_{\mathcal{G}}$ be a corresponding program synthesized by the inductive lifting procedure Lift (Algorithm 1). Let $\mathcal{M}_{\text{BTC}}$ and $\mathcal{M}_{\mathcal{G}}$ denote the operational semantics of Bitcoin script and the $\mathcal{G}$ language respectively, formally defined by their respective execution models (Section 4). Suppose for all input states s, executing P_{BTC} under $\mathcal{M}_{\text{BTC}}$ and executing $P_{\mathcal{G}}$ under $\mathcal{M}_{\mathcal{G}}$ produce behaviorally equivalent outcomes (i.e., identical observable final stack states and outputs):

$$\forall s. \mathcal{M}_{\text{BTC}}[P_{\text{BTC}}](s) \approx \mathcal{M}_{\mathcal{G}}[P_{\mathcal{G}}](s),$$

then verifying a correctness specification Φ at the $\mathcal{G}$ level implies correctness at the Bitcoin script level:

$$\mathcal{M}_{\mathcal{G}} \models P_{\mathcal{G}} : \Phi \implies \mathcal{M}_{\text{BTC}} \models P_{\text{BTC}} : \Phi,$$

assuming the equivalence checker used by the Lift is sound and complete within the considered input domain.

Proof Sketch. We first clarify the involved semantics: $\mathcal{M}_{\text{BTC}}$ is the formal operational semantics that precisely describes execution rules for Bitcoin script operations (Section 4.3), while $\mathcal{M}_{\mathcal{G}}$ denotes the formal semantics defined for our register-style intermediate language $\mathcal{G}$ (Section 4.2 and Section 4.3).

The inductive synthesis procedure Lift produces a $\mathcal{G}$ program $P_{\mathcal{G}}$ from P_{BTC} via iterative synthesis (Algorithm 1). Each step involves symbolic equivalence checking, which formally verifies that both programs produce identical observable behaviors for all inputs. This equivalence checking is assumed sound and complete, ensuring that no behaviorally inequivalent transformations are admitted.

Given behavioral equivalence, correctness verification conducted using $\mathcal{M}_{\mathcal{G}}$ on $P_{\mathcal{G}}$ directly transfers correctness claims to the original Bitcoin script P_{BTC} executed under $\mathcal{M}_{\text{BTC}}$. Specifically, if a correctness specification Φ (such as safety properties or functional correctness) is satisfied by $P_{\mathcal{G}}$, then it must also hold for P_{BTC}, due to the exact correspondence in observable behaviors as enforced by the equivalence checker.

Therefore, verification at the $\mathcal{G}$ level is soundly reflected at the Bitcoin script level. □

6 Implementation

We have implemented bitguard in Rosette [Torlak and Bodik 2014] with a back-end constraint solver (Bitwuzla [Niemetz and Preiner 2023] version 0.4.0). The total codebase comprises 2,574 lines of 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 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:

- $\mathcal{N}$: The total number of bits of the BigInt.
- $\mathcal{L}$: The number of bits per limb.
- l: The base name for each limb, with l_i 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:

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

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:

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

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:

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

We then apply the following constraint to the highest limb s_n: $s_n[\text{sign}] = 0$, 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 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.

Loop invariant templates. Similar to previous work on loop invariant inference, we provide a set of templates as domain-specific knowledge to guide and prioritize the search. Below is a subset of representative templates for constructing loop invariants:

- A symbolic variable is zero: $v = 0$.
- A variable is binary: $v = 0 \vee v = 1$.
- A variable representing a 29-bit limb is within its valid range: $0 \leq v < 2^{29}$.
- The i-th element of the stack is equal to a symbolic variable v: $\mu[i] = v$.

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 104 verification tasks from the three 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:

- **Bitvm bridge**[1] (or **BVM** for short) implements BitVM2 [Linus et al. 2024], 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**[2] (or **BSV** for short) implements a circle plonk [Gabizon et al. 2019] verifier in Bitcoin script. It also comes with reusable cryptographic components written in Bitcoin script.
- **Arithmetic over the M31 or BabyBear field in Bitcoin script**[3] (or **MBB** for short) implements efficient M31 and BabyBear field arithmetic in Bitcoin script, providing reusable building blocks for zero-knowledge and cryptographic protocols on Bitcoin.

Among our 104 benchmarks, 63 benchmarks are from BVM, 11 from BSV, and 30 from MBB. Each benchmark has on average 525,411 lines of code, with a maximum of 5,780,711 Bitcoin script opcodes. 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.

Specification and Properties. bitguard verifies functional correctness of each benchmark (i.e., ensuring that computations produce expected results). Each benchmark follows the standard definitions of cryptographic operations, whose documentation is generally available in various prevailing tools (e.g., circom-pairing[4]). We construct the specification according to the documentation using bitguard 's DSL.

[1] https://github.com/BitVM/BitVM/tree/main/bitvm/src
[2] https://github.com/Bitcoin-Wildlife-Sanctuary/bitcoin-circle-stark
[3] https://github.com/BitVM/rust-bitcoin-m31-or-babybear
[4] https://github.com/yi-sun/circom-pairing/tree/master/docs

	Total	Avg. Time	Solved	Safe (✔)	Unsafe (✕)	Unknown (?)
BVM	63	37.05s	50 (79%)	49 (78%)	1 (2%)	13 (21%)
BSV	11	3.49s	11 (100%)	11 (100%)	0 (0%)	0 (0%)
MBB	30	2.62s	30 (100%)	26 (87%)	4 (13%)	0 (0%)
Overall	104	23.57s	91 (88%)	86 (83%)	5 (5%)	13 (12%)

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

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 [Barrett et al. 1998] and leverages Bitwuzla [Niemetz and Preiner 2023] 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 ("✕"), 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. Since predicates are generated by unwinding the $\mathcal{G}$ grammar to a fixed bound, some counterexamples may be spurious, leading to incorrect "unsafe" conclusions. To address this, we integrate a CEGAR-based refinement process, which refines invariants by eliminating spurious counterexamples. 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 104 benchmarks, bitguard solves 91 (88%) of them, with 86 (83%) proven safe (✔) and 5 (5%) having counterexamples found, i.e., proven unsafe (✕). bitguard takes an average of 23.57s to solve a benchmark. Only 13 (12%) 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. Two of the benchmarks have user-provided annotations. 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. 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

Benchmark	LOC	Result	Time (s)
(BVM) bigint/std			
zip	36	✔	2.03
copy	18	✔	2.02
roll	36	✔	2.02
is_zero_ke	37	✔	2.16
is_one_ke	38	✔	2.21
totalstack	9	✔	1.99
fromaltstack	18	✔	2.01
is_negative	4	✔	2.16
is_positive	4	✘	2.16
resize	12	✔	1.92
overall	19	100%	2.07
(BVM) bigint/add			
add	173	✔	3.58
double_allow_overflow_ke	134	✔	3.58
double_prevent_overflow	133	✔	3.54
lshift_prevent_overflow	3,713	✔	5.54
add_ref_with_top	164	✔	4.03
overall	863	100%	4.05
(BVM) bigint/bits			
convert_to_be_bits	3,297	✔	4.38
convert_to_le_bits	3,297	✔	4.35
convert_to_be_bits_ta	3,081	✔	4.34
convert_to_le_bits_ta	3,535	✔	4.36
limb_from_bytes	121	✔	2.14
overall	2,666	100%	3.91
(BVM) bigint/inv			
div2	4,157	✔	6.26
div2rem	4,156	✔	6.48
div3	5,057	✔	2.29
div3rem	5,056	✔	2.40
div3_toaltstack	5,295	✔	3.63
inv_stage1	4,992,182	?	TO
overall	835,983	83%	4.21
(BVM) bn254/curves			
push_generator	27	✔	1.95
push_zero	15	✔	1.96
is_zero_ke	37	✔	2.17
add	2,180,126	?	TO
double	947,166	?	TO
equalverify	1,089,148	?	TO
into_affine	5,780,711	?	TO
overall	1,428,176	43%	2.03
(BVM) bigint/mul			
mul	102,932	✔	452.10
mul_ke	102,952	✔	454.80
mul_toaltstack	103,170	✔	455.31
u29_mul_toaltstack	1,065	✔	35.19
u29_mul_ke	837	✔	34.84
u29_mul	827	✔	33.57
overall	51,964	100%	244.30
(BVM) bigint/sub			
sub	180	✔	2.14
overall	180	100%	2.14
(BSV) utils			
limb_to_le_bits	376	✔	2.48
ltbbt_exc_low2b	351	✔	2.13
ltbbt_common	349	✔	2.40
qm31_reverse	3	✔	2.00
ltbbt_exc_low1b	351	✔	2.09
dup_mv_g	64	✔	2.05
mv_from_bottom_g	96	✔	2.05
cta_top_item_first_in_g	28	✔	2.00
overall	202	100%	2.15

Benchmark	LOC	Result	Time (s)
(BSV) folding			
check_0_or_1	8	✔	2.11
decompose_positions_g	436	✔	11.95
skip_one_and_ext_bits_g	361	✔	7.14
overall	268	100%	7.07
(BVM) bn254/fp254impl			
div2	4,547	✔	4.85
div3	5,845	✔	95.41
div3_toaltstack	6,083	✔	97.23
convert_to_be_u4	4,007	✔	28.13
convert_to_be_bits	3,297	✔	4.22
convert_to_be_bits_ta	3,081	✔	4.28
convert_to_le_bits	3,297	✔	4.31
convert_to_le_bits_ta	3,535	✔	4.39
sub	396	✔	5.44
sub_fq2	792	✔	8.90
sub_fq6	2,376	✔	27.68
sub_fq	396	✔	5.35
double	350	✔	2.30
inv	5,235,924	?	TO
mul_by_constant	101,871	?	TO
square	133,523	?	TO
mul	136,960	?	TO
mul_bucket	71,852	?	TO
decode_mtg	63,322	?	TO
convert_to_be_bytes	67,363	?	TO
mul_by_constant_bucket	67,745	?	TO
overall	281,741	62%	22.50
(BVM) bigint/cmp			
equalverify	45	✔	2.13
lessthanorequal	183	✔	2.13
overall	114	100%	2.13
(MBB) babybear			
u31_adjust	7	✔	2.01
u31_double	11	✔	2.34
u31_to_v31	2	✔	2.15
u31_add	10	✔	2.31
u31_neg	3	✘	2.40
u31_add_v31	8	✔	2.12
v31_neg	3	✘	2.42
v31_add_u31	8	✔	2.13
v31_adjust	7	✔	1.98
u31_sub	8	✔	2.18
v31_double	11	✔	2.31
v31_to_u31	2	✔	2.13
u31_to_bits	270	✔	8.37
v31_sub	8	✔	2.16
v31_add	10	✔	2.19
overall	25	100%	2.61
(MBB) m31			
u31_adjust	7	✔	2.00
u31_double	11	✔	2.39
u31_to_v31	2	✔	2.20
u31_add	10	✔	2.33
u31_neg	3	✘	2.50
u31_add_v31	8	✔	2.14
v31_neg	3	✘	2.47
v31_add_u31	8	✔	2.13
v31_adjust	7	✔	1.96
u31_sub	8	✔	2.11
v31_double	11	✔	2.34
v31_to_u31	2	✔	2.10
u31_to_bits	270	✔	8.29
v31_sub	8	✔	2.11
v31_add	10	✔	2.19
overall	25	100%	2.62

Table 3. Statistics and breakdown of bitguard'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.

generate non-linear constraints. However, even though inv_stage1 contains only 1 loop, bitguard fails to solve it due to the complicated loop invariants.

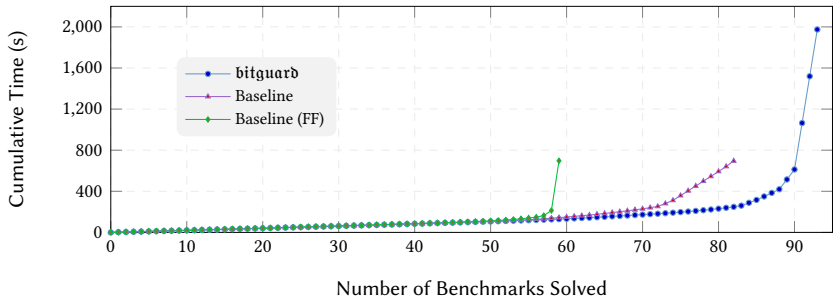


Fig. 8. A comparison between bitguard and its baselines without transpilation, where x-axis denotes the total number of benchmarks solverd, and y-axis denotes the cumulative time spent in seconds.

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 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 (91 out of 104, i.e., 88%) of benchmarks with a 23.57s averaged solving time, where synthesis of snippets takes an average of 1.49s (around 6% of total 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 [Klomp and Bracciali 2018]. 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.

A subset of benchmarks (15.4%, 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 [Pailoor et al. 2023]. To explore whether a finite field solver could improve performance, we introduce a second ablative version, Baseline (FF). This version uses cvc5 [Barbosa et al. 2022] with specialized finite field theory [Ozdemir 2022] (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 8 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, Baseline (FF) underperforms compared to both bitguard and Baseline. 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, bitguard demonstrates a clear advantage in 20.19% of benchmarks, thanks to the high-level DSL and the synthesis procedure discussed in Section 5.1. Baseline (FF) initially performs similarly to bitguard for the first 50 benchmarks but falls behind as more benchmarks are added, with bitguard ultimately solving 18.27% more benchmarks.

A closer analysis reveals that bitguard actually solves some of the largest and most challenging benchmarks that other variants cannot, such as div3 and mul. Notably, the div3 series — including bigint's div3, div3rem and bn254's div3 — are among the most fundamental operations in modern cryptographic protocols (e.g., bigint and bn254). BitGuard's experimental results demonstrate its ability to efficiently handle these critical benchmarks, achieving substantial performance gains rather than merely incremental improvements on easier ones.

In addition to the ablation configurations discussed above, we separately report the synthesis time of the tool, denoted by bitguard (synth. only). The synthesis time consistently accounts for only a small fraction of the overall verification time across all benchmarks. In most cases, the synthesis step takes less than 5% - 9% of the total, while the majority of the computation is consumed by symbolic evaluation and constraint solving. This observation confirms that the cost of automated DSL lifting does not constitute a practical bottleneck, and highlights the scalability of our approach even for large, real-world BitVM programs.

Result for RQ2: bitguard performs significantly better than its ablative versions, with notable efficiency gains in 20.19% - 32.69% of cases. Both the synthesis phase and the overall workflow demonstrate strong scalability across benchmarks. These results highlight the effectiveness and scalability of the framework's design, answering RQ2 in a positive way.

7.3 Detecting Previously Unknown Vulnerabilities (RQ3)

As shown in Table 2, bitguard has identified **5** previously unknown vulnerabilities, with all of them confirmed by the developers. In this section, we elaborate in more detail about the zero-days found. In particular, those vulnerabilities can be grouped into two categories:

Buggy zero handling in positive number detection. 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 9(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 9(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. Our tool uncovered one zero-day vulnerability in this category.

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


    { Self::HEAD_OFFSET >> 1 }
    OP_LESSTHAN

  }
}
```

(a) the buggy snippet

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

(b) the fixed snippet

Fig. 9. An example code snippet demonstrating a bug in BitVM's implementation (a) and its fixed version (b).

Canonical encoding violations in field arithmetic. In addition to the `is_positive` zero-day, we uncovered four more zero-days in **MBB**. It encodes numbers in two related formats: `u31`, which breaks each value into unsigned 31-bit limbs and then reassembles them modulo a prime, and `v31`, which does the same but treats the top bit of each limb as a sign bit for signed arithmetic. Both of the `u31` negation routines (`u31_neg` and its subtraction-based variant) mistakenly folded `0` into the `MOD` (and thus gave `0` two distinct representations), and likewise both of the `v31` negation routines (`v31_neg` and its subtraction variant) collapsed the special value `-MOD` back into `0`. These misencodings violate the fundamental invariant that each field element has exactly one canonical representation, which in turn can let an attacker bypass zero-checks or range-checks that assume uniqueness and trigger undefined or insecure behavior. Our tool uncovered four zero-day vulnerabilities in this category.

Result for RQ3: `bitguard` detected **5** previously unknown vulnerabilities in widely-used Bitcoin scripts, which could allow invalid proofs to be mistakenly accepted as valid. These results highlight the critical role of `bitguard`'s verification design in identifying logical flaws from modern programmable Bitcoin systems.

8 Related work

Recent work has addressed formal methods for cryptography, zero-knowledge proof systems, and smart contract verification. We briefly review the most relevant developments in these areas as they relate to our approach.

Formal methods for cryptography. There is extensive research on applying formal verification techniques to cryptographic protocols. For example, Corin et al.[Corin and den Hartog 2005] utilized a variant of probabilistic Hoare logic to verify the security of ElGamal, while Gagne et al.[Gagné et al. 2013] applied similar methods to analyze the security of CBC-based MACs, PMAC, and HMAC. Tiwari et al. [Tiwari et al. 2015] employed component-based program synthesis to automatically generate padding-based encryption schemes and block cipher modes of operation. EasyCrypt [Barthe et al. 2013] 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.[Almeida et al. 2010] developed a certifying compiler for Σ-protocols, which includes zk-SNARKs, using Isabelle/HOL [Nipkow et al. 2002] for formal correctness proofs. Sidorenco et al.[Sidorenco et al. 2021] 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. [Hader 2022] 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. [Ozdemir et al. 2023] recently proposed a finite field solver that does not scale well in our benchmarks due to too many complex constraints. Finally, Coda [Liu et al. 2024] 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 [Dahlgren 2022] 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 [Wen et al. 2024] 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)[Nelson and Oppen 1980] has become an essential tool for symbolic reasoning, driven by the availability of practical, high-performance solvers like Z3[de Moura and Bjørner 2008], CVC4[Barrett et al. 2011], and Gurobi[Gurobi Optimization 2019]. The programming languages community has extensively explored the use of solvers for both verification and synthesis [Leino 2010; Schkufza et al. 2013; Solar-Lezama 2008]. 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)[Torlak and Bodik 2014; Uhler and Dave 2014] automatically generate these constraints through symbolic compilation. One example is the Rosette framework[Torlak and Bodik 2014], 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

This work tackles the correctness gap at the heart of *programmable Bitcoin*. We presented bitguard, a synthesis-aided verifier that lifts raw Bitcoin script into a register-style DSL, detects the repetitive slices that emulate batch operations, and replaces them with axiomatized higher-order combinators. A counter-example-guided inductive-synthesis engine then proves each lifted fragment equivalent to its original script, after which off-the-shelf SMT solvers can discharge the remaining obligations in seconds rather than hours. The approach eliminates the need for developers to reason directly about megabytes of stack gymnastics while retaining fidelity to Bitcoin's execution model.

An evaluation on seventy-four real-world artefacts, including full BitVM2 prover–verifier pairs, covenant templates, and cryptographic sub-routines, shows that bitguard verifies 88% of the benchmarks in an average of 23.57 seconds and uncovers **5** previously unknown vulnerabilities. These results demonstrate that automated lifting, axiomatized batch combinators, and inductive

synthesis together provide a practical path to rigorous assurance for the growing ecosystem of Bitcoin-centric DeFi and cross-chain applications.

References

José Almeida, Endre Bangerter, Manuel Barbosa, Stephan Krenn, Ahmad-Reza Sadeghi, and Thomas Schneider. A certifying compiler for zero-knowledge proofs of knowledge based on sigma-protocols. volume 6345, pages 151–167, 09 2010. doi: 10.1007/978-3-642-15497-3_10.

Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22109-5. URL http://dl.acm.org/citation.cfm?id=2032305.2032319.

Clark W Barrett, David L Dill, and Jeremy 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.

Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *FOSAD*, 2013.

Rob Behnke. Explained: The pNetwork Hack (September 2021), 2021. URL https://www.halborn.com/blog/post/explained-the-pnetwork-hack-september-2021. Accessed 28 Jun 2025.

Rob Behnke. Explained: The Sovryn Hack (October 2022), 2022. URL https://www.halborn.com/blog/post/explained-the-sovryn-hack-october-2022. Accessed 28 Jun 2025.

Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars 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, August 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson.

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

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In Hans-Juergen Boehm and Cormac 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.

Cointelegraph. Alex Lab points to Lazarus Group after last month's $4M exploit, 2024. URL https://cointelegraph.com/news/bitcoin-layer-2-alex-lab-may-exploit-lazrus-group-north-korea. Accessed 28 Jun 2025.

Cointelegraph. Bitcoin DeFi platform Alex Protocol loses $8.3M to exploit, 2025. URL https://cointelegraph.com/news/bitcoin-defi-platform-alex-protocol-loses-8-3m-to-exploit. Accessed 28 Jun 2025.

Ricardo Corin and Jerry den Hartog. A probabilistic hoare-style logic for game-based cryptographic proofs (extended version), 2005. URL http://eprint.iacr.org/2005/467. To appear in ICALP 2006 Track C corin@cs.utwente.nl 13264 received 23 Dec 2005, last revised 26 Apr 2006.

Fredrick Dahlgren. It pays to be circomspect. https://blog.trailofbits.com/2022/09/15/it-pays-to-be-circomspect/, 09 2022.

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

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 422–436, New York, NY, USA, 2017. Association for Computing Machinery.

Cormac Flanagan and K Rustan 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.

Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019.

Martin Gagné, Pascal Lafourcade, and Yassine Lakhnech. Automated security proofs for almost-universal hash for mac verification. Cryptology ePrint Archive, Paper 2013/407, 2013. URL https://eprint.iacr.org/2013/407. https://eprint.iacr.org/2013/407.

LLC Gurobi Optimization. Gurobi optimizer reference manual, 2019. URL http://www.gurobi.com.

Thomas Hader. Non-linear smt-reasoning over finite fields, 2022.

Ethan Heilman and Armin Sabouri. BIP-420: OP_CAT. https://github.com/bip420/bip420, 2023. Draft Bitcoin Improvement Proposal, accessed 28 Jun 2025.

Hiro Systems PBC. Stacks (stx). https://stacks.co, 2025. Bitcoin Layer 2 enabling smart contracts and DeFi using Clarity.

IOV Labs. Rootstock (rsk). https://rootstock.io, 2025. EVM-compatible Bitcoin sidechain secured by BTC hash power.

Rick Klomp and Andrea Bracciali. On symbolic verification of bitcoin's script language. In Joaquin Garcia-Alfaro, Jordi Herrera-Joancomartí, Giovanni Livraga, and Ruben Rios, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 38–56, Cham, 2018. Springer International Publishing.

K. Rustan 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. ISBN 3-642-17510-4, 978-3-642-17510-7. URL http://dl.acm.org/citation.cfm?id=1939141.1939161.

Robin Linus, Lukas Aumayr, Alexei Zamyatin, Andrea Pelosi, Zeta Avarikioti, and Matteo Maffei. BitVM2: Bridging bitcoin to second layers, August 2024.

Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Isil Dillig, and Yu 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. doi: 10.1109/SP54263.2024.00078. URL https://doi.org/10.1109/SP54263.2024.00078.

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

Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980. ISSN 0004-5411. doi: 10.1145/322186.322198. URL http://doi.acm.org/10.1145/322186.322198.

Aina Niemetz and Mathias Preiner. Bitwuzla. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 3–17, Cham, 2023. Springer Nature Switzerland.

Tobias Nipkow, Markus Wenzel, and Lawrence Charles Paulson. Isabelle/hol: A proof assistant for higher-order logic. 2002.

Alex Ozdemir. Cvc5-ff. https://github.com/alex-ozdemir/CVC4/tree/ff, 2022.

Alex Ozdemir, Gereon Kremer, Cesare Tinelli, and Clark W. Barrett. Satisfiability modulo finite fields. In Constantin Enea and Akash 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. doi: 10.1007/978-3-031-37703-7_8. URL https://doi.org/10.1007/978-3-031-37703-7_8.

Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Gaffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Isil Dillig. Automated detection of underconstrained circuits for zero-knowledge proofs. Cryptology ePrint Archive, Paper 2023/512, 2023. URL https://eprint.iacr.org/2023/512. https://eprint.iacr.org/2023/512.

Jeremy Rubin. BIP-119: OP_CHECKTEMPLATEVERIFY. https://github.com/bitcoin/bips/blob/master/bip-0119.mediawiki, 2020. Bitcoin Improvement Proposal, accessed 28 Jun 2025.

Eric Schkufza, Rahul Sharma, and Alex 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. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451150. URL http://doi.acm.org/10.1145/2451116.2451150.

Nikolaj Sidorenco, Sabine Oechsner, and Bas 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. doi: 10.1109/CSF51468.2021.00050.

Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.

Sovryn Community. Sovryn. https://www.sovryn.app, 2025. Bitcoin-native DeFi protocol built on RSK.

Ashish Tiwari, Adria Gascon, and Bruno 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. doi: 10.1007/978-3-319-21401-6_33. URL http://dx.doi.org/10.1007/978-3-319-21401-6_33.

Emina Torlak and Rastislav 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.

Richard Uhler and Nirav 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. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660208. URL http://doi.acm.org/10.1145/2660193.2660208.

Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. Practical security analysis of zero-knowledge proof circuits. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024. URL https://www.usenix.org/conference/usenixsecurity24/presentation/wen.