

BitVM: Quasi-Turing Complete Computation on Bitcoin

Lukas Aumayr 

University of Edinburgh, UK
Common Prefix, Edinburgh, UK

Robin Linus Woll 

Stanford University, CA, USA
ZeroSync, Zug, Switzerland

Andrea Pelosi 

TU Wien, Austria

Alexei Zamyatin 

BOB, London, UK

Zeta Avarikioti 

TU Wien, Austria
Common Prefix, Vienna, Austria

Matteo Maffei 

TU Wien, Austria

Christos Stefo 

TU Wien, Austria

Abstract

A long-standing question in the blockchain community is which class of computations is efficiently expressible in cryptocurrencies with limited scripting languages, such as Bitcoin Script. Such languages expose a reduced trusted computing base, thereby being less prone to hacks and vulnerabilities, but have long been believed to support only limited classes of payments.

In this work, we confute this long-standing belief by showing for the first time that arbitrary computations can be encoded in today's Bitcoin Script without introducing any language modification or additional security assumptions, such as trusted hardware, trusted parties, or committees with an honest majority. We present BitVM, a two-party protocol that realizes a generic virtual machine by combining cryptographic primitives and economic incentives. We conduct a formal analysis of BitVM, characterizing its functionality, system assumptions, and security properties. We further demonstrate the practicality of our approach by implementing a prototype and performing an experimental evaluation: in the optimistic case (i.e., when parties agree), our protocol requires just three on-chain transactions, whereas in the pessimistic case, the number of transactions grows logarithmically with the size of the virtual machine. We exemplify the deployment potential of BitVM by building a Bitcoin-sidechain bridge application. This work not only solves a long-standing theoretical problem, but it also promises a strong practical impact, enabling the development of complex applications in Bitcoin.

2012 ACM Subject Classification Security and privacy → Security protocols; Security and privacy → Distributed systems security

Keywords and phrases Bitcoin, quasi-Turing completeness, off-chain protocols, bridge

Digital Object Identifier 10.4230/LIPIcs.AFT.2026.1

Related Version *Full Version*: <https://eprint.iacr.org/2024/1995.pdf> [5]

Supplementary Material *Software (source code)*: <https://github.com/BitVM/bitvm-js> [2]
archived at `swh:1:dir:fd9022956a1774f59cf49dacf6e87d32f90863aa`

Funding This work was partially supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 101141432-BlockSec), by the Austrian Science Fund (FWF) through the SFB SpyCode project (F8510-N and F8512-N), by the Vienna Science and Technology Fund (WWTF) through the projects 10.47379/ICT25056 and 10.47379/ICT22045, and by Input Output (<https://iohk.io>) through their funding of the Edinburgh Blockchain Technology Lab.



© Lukas Aumayr, Zeta Avarikioti, Robin Linus Woll, Matteo Maffei, Andrea Pelosi, Christos Stefo, and Alexei Zamyatin;

licensed under Creative Commons License CC-BY 4.0

8th Conference on Advances in Financial Technologies (AFT 2026).

Editors: Aggelos Kiayias and Maria Kyropoulou; Article No. 1; pp. 1:1–1:23



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Smart contracts are a foundational component of modern blockchain systems, enabling decentralized applications (dApps) and programmable money without the need for trusted intermediaries. These self-executing programs have unlocked a wide range of use cases, spanning finance, governance, and supply chain management, by enforcing agreement logic directly on-chain.

While some blockchains, such as Ethereum, support quasi-Turing complete¹ execution through bytecode languages like the Ethereum Virtual Machine (EVM), others, most notably Bitcoin, opt for a minimalist approach. Bitcoin Script deliberately limits expressiveness to minimize complexity and attack surface, trading expressiveness for security and stability.

This trade-off has given rise to a long-standing open question: *Can general-purpose computation be supported on Bitcoin, using only its existing scripting language and consensus rules?* Unlocking such expressiveness could expand Bitcoin’s utility for dApps and decentralized finance (DeFi), while preserving its conservative security model.

The prevailing belief in the blockchain community is that Bitcoin Script cannot support general-purpose computation in practice. This belief stems from several structural limitations of the language: it is stateless, lacks loops and recursion, and enforces strict constraints on script and transaction size. These properties make it well-suited for simple functionalities, such as multisignature payments or hashed timelock contracts, but appear to rule out the execution of complex logic on-chain.

Several proposals have attempted to overcome these constraints, either by extending Bitcoin with new opcodes (e.g., covenants [12]), encoding computations into low-level counter machines [13], or relying on trusted execution environments [18, 24] and oracles [33, 19]. However, these approaches either require consensus changes, incur prohibitive on-chain costs, or compromise Bitcoin’s trust model. As a result, it has long been assumed that supporting arbitrary computation on Bitcoin would require trade-offs incompatible with its conservative design philosophy.

Contributions. In this work, we challenge this long-standing belief by showing that arbitrary (bounded) computations can be executed securely on Bitcoin in a practical manner. We introduce BitVM², a two-party protocol that enables expressive, verifiable off-chain computation using only existing Bitcoin features. BitVM requires no consensus changes, no new opcodes, and no trusted hardware or oracles.

The core idea behind BitVM is to shift computation off-chain while retaining on-chain verifiability through a fraud-proof mechanism. Specifically, a prover submits a claim about the output of a bounded computation. The verifier can either accept the result or, in case of disagreement, initiate an interactive dispute protocol. This protocol relies on a custom virtual machine that encodes computation as an execution trace, enabling the parties to identify a point of disagreement and verify a single step of computation using Bitcoin Script.

When both parties agree, the entire process completes with just three on-chain transactions. In case of dispute, BitVM guarantees that verification incurs only a logarithmic number of additional transactions. This makes BitVM both practical and expressive, enabling Bitcoin to support advanced smart contract functionality without compromising its trust model or requiring protocol changes.

¹ This term is adopted in the blockchain community to indicate Turing-complete languages that enforce termination by bounding the execution (e.g., via gas consumption in Ethereum) [35].

² This work extends and formalizes the original BitVM design, which was conceptualized and developed by Robin Linus [29].

To illustrate how BitVM operates in practice, consider a simple two-party wager: a prover claims to have solved a chess puzzle, and a verifier bets against them. Both parties lock funds into a BitVM contract that verifies the claimed solution. If the verifier agrees, the outcome is settled off-chain. If not, they can initiate an on-chain dispute. The contract then executes the verification step by step, and settles the funds accordingly, using only Bitcoin Script to enforce the outcome. While conceptually simple, this example demonstrates the core functionality of BitVM: enabling trustless agreement on arbitrary computation, with minimal on-chain footprint in the absence of disputes.

The same mechanism underpins more complex applications. A key example, which we outline in this work, is a *trust-minimized bridge between Bitcoin and a sidechain*. In typical designs, a committee of operators handles redemptions from the sidechain to Bitcoin, with security guaranteed only under an honest-majority assumption (t -of- n , where $t > n/2$), e.g., [9, 1]. With BitVM, this trust requirement is minimized to *existential honesty*: security holds as long as *at least one* operator behaves honestly ($t = 1$). Any operator can front coins to a user on Bitcoin and later claim reimbursement by proving correctness; if another committee member disputes the claim, they can initiate a BitVM instance to verify or refute it on-chain. This approach *reduces the trust assumption from honest majority of the bridge committee to existential honesty*, while remaining fully compatible with Bitcoin’s existing scripting model. Crucially, such a bridge serves as a foundational building block for a broader ecosystem of decentralized applications: once assets can move between Bitcoin and external execution layers securely and trustlessly, DeFi protocols, from decentralized exchanges to lending platforms, can operate on top of Bitcoin without compromising its security guarantees.

The appeal of the BitVM paradigm is witnessed by the massive investments of the industry (more than 150M USD³) and the number of BitVM-based protocols already in development [23, 38, 17, 4].

This paper makes the following contributions:

- We present BitVM, the first protocol to encode quasi-Turing complete computations in Bitcoin Script, requiring no consensus changes or trusted third parties (Section 5).
- We provide a formal analysis of BitVM, characterizing its functionality, system assumptions, and security properties (Section 6).
- To evaluate BitVM’s performance, we implement a prototype of BitVM in JavaScript.⁴ Optimistic execution completes in three on-chain transactions, costing approximately 5,832 satoshis⁵, equivalent to \$4.26 (as of May 2026). In case of disputes, the on-chain footprint grows logarithmically with the computation size. For a virtual machine with 2^{32} memory cells and steps – comparable to a high-end 1990s workstation – settlement requires up to 81 transactions, at a cost of approximately 732,000 satoshis, equivalent to \$534 (Section 7).
- We demonstrate the capabilities of BitVM by constructing a trust-minimized bridge protocol between Bitcoin and a sidechain, reducing the traditional honest-majority assumption to that of a single honest operator (Section 8).

Related work. Several works have attempted to overcome the limited expressiveness of Bitcoin Script and enable more complex smart contracts by combining UTXOs and scripts, effectively splitting functionality across multiple transactions. BitML [14] provides a high-level, domain-specific language for smart contracts with an associated compiler that produces

³ <https://cryptorank.io>

⁴ The prototype is available at [2].

⁵ A *satoshi* is a fraction of a bitcoin, i.e., $1\text{sat} = 10^{-8}\text{B}$.

Bitcoin transactions, illustrating Bitcoin’s potential for intricate smart contract designs [11]. These methods, however, incur substantial on-chain costs, as compiled programs often result in numerous large transactions that must ultimately be recorded on-chain, and they support only a restricted class of contracts.

To mitigate these costs, some approaches leverage Trusted Execution Environments (TEEs). FastKitten [18] facilitates off-chain computation within a secure hardware enclave, but relies on collateral, rational adversaries, trusted TEE operators, and a limited contract duration. POSE [24] improves upon FastKitten by removing collateral requirements and time constraints, and by enhancing privacy, but it continues to rely on trusted TEE hardware.

A different approach uses Hashed Timelock Contracts (HTLCs) to shift computation off-chain by encoding outcomes in preimages of hash functions [8], similar in spirit to *state channels* on Ethereum [22, 20]. This model underpins constructions such as Discreet Log Contracts (DLCs) [19] and oracle-based conditional payments [33], which depend on (semi-)trusted oracles to attest to specific events. A key limitation of these approaches is that all possible outcomes must be known and encoded in advance. Consequently, they cannot support applications like the chess puzzle or the bridge example, where the correct outcome, such as the solution to a puzzle or the identity of the party holding funds on the sidechain, may not be known a priori to any participant.

Unlike prior works, BitVM enables quasi-Turing complete computation on Bitcoin without consensus changes or relying on external trust assumptions, such as TEEs and semi-trusted oracles. It is the first trustless protocol to allow arbitrary, bounded computation on Bitcoin, while incurring logarithmic on-chain cost in the worst case, unlocking a range of potential applications. Table 1 situates BitVM among prior Bitcoin smart contract approaches.

■ **Table 1** Comparison of Bitcoin-based smart contract approaches. n denotes the upper bound on computational steps, and QT refers to Quasi-Turing completeness.

Approach	Expressiveness	Extra Assumptions	On-chain cost
BitML [14, 11]	QT	None	$O(n)$
TEEs [18, 24]	QT	TEE	$O(n)$
Gen. Channels [8]	Bitcoin	None	$O(1)$
Oracles [19, 33]	QT	Trusted oracle	$O(1)$
BitVM (our work)	QT	None	$O(\log(n))$

Follow-up work [30], informally referred to as BitVM2, targets bridges between Bitcoin and Layer-2 systems. It compiles a SNARK verifier into Bitcoin Script, splits it across transactions, and commits to intermediary states on-chain. This supports permissionless dispute resolution but incurs on-chain cost linear in the verifier size, requiring a transaction that fills a 4MB Bitcoin block in case of disputes. Our BitVM is instead better suited to permissioned settings and resolves disputes at logarithmic on-chain cost, with formal security guarantees. Beyond the asymptotic gap, it is also cheaper in practice: Table 2 shows on-chain costs an order of magnitude below BitVM2 in both the optimistic and dispute cases, where for a fair comparison we size BitVM at 2^{32} memory cells, sufficient to encode the SNARK verifier. Finally, BitVM runs RISC-V programs through off-the-shelf compilers, whereas BitVM2 requires compiling the verifier (around 4GB) into Bitcoin Script, a cumbersome and error-prone task.

A recent development, informally termed BitVM3 [40], pushes this paradigm further by trading on-chain cost for off-chain computation: it encodes the BitVM2 SNARK verifier as a garbled circuit. The operator garbles the verifier off-chain and commits to it during setup; a challenger recovers a fraud-proof secret off-chain iff the claim is incorrect, so a dispute is settled by revealing a single secret on-chain. BitPriv [6] and BABE [26] build on this

paradigm, the former adding input privacy and the latter reducing verification costs under a permissioned challenger set. The trade-off is steep off-chain cost: each garbled circuit reaches 40GB, yielding terabytes of storage and communication overhead.

Table 2 summarizes the three BitVM paradigms. BitVM is the first to introduce this paradigm; BitVM2 and BitVM3 are follow-ups that trade it off differently, the former for permissionless disputes at linear on-chain cost, the latter for lower on-chain cost at the expense of heavy off-chain garbling.

■ **Table 2** On-chain and off-chain costs comparison between BitVM paradigms.

Approach	Optimistic (\$)	Dispute (\$)	Optimistic (vB)	Dispute (vB)	Off-chain cost
BitVM (our work)	4.26	534	1.9k	244k	Low
BitVM2 [31]	38.7	11,680	16.8k	496.7k	Low
BitVM3 [40]	5.26	5.48	2.4k	2.5k	High

Finally, we note that the optimistic verification paradigm behind BitVM is inspired by optimistic rollup protocols [27]. While optimistic rollups are prominent in the Ethereum ecosystem, adapting this principle to Bitcoin, which offers limited scripting flexibility and strict block size constraints, remained an open question until now.

2 Model

BitVM is a protocol between two mutually distrusting parties, the prover P and the verifier V , designed to enable P to prove on the Bitcoin blockchain that the outcome of a pre-agreed computation with V was performed correctly. Concretely, for an agreed-upon Turing-complete program Π , a BitVM instance secures collateral from both parties and it enables P to enforce a transaction on-chain based on the outcome $\Pi(x)$ for a specific input x . In other words, $\Pi(x)$ dictates the payout of the funds within the BitVM instance, typically allocating them to P and V ⁶. If P or V stop collaborating during protocol execution, after a designated period all the funds are allocated to the other party.

2.1 System model

We assume time advances in discrete rounds $(1, 2, \dots)$. Protocol participants run in probabilistic polynomial time (PPT) in the security parameter κ . We assume synchronous communication, i.e., messages sent between parties arrive at the beginning of the next round, as well as authenticated communication channels. Our protocol employs a hash function modeled as a random oracle $H : \{0, 1\}^* \rightarrow \{0, 1\}^\kappa$ which maps an input of arbitrary length to a fixed κ -sized output. Moreover, our protocol builds upon a distributed ledger protocol (e.g., [25, 39, 7]).

► **Definition 1** (Distributed Ledger Protocol). *A distributed ledger protocol is an interactive Turing machine exposing the following functionality on each party.*

- *execute()*: executes one protocol round and enables the machine to communicate with the network, invoked by the environment in every round;
- *write(tx)*: takes as input a transaction from the environment;
- *read()*: outputs a finite, ordered sequence of transactions, known as transaction ledger L .

⁶ Note that P and V can also agree to allocate the funds to a third party or, more generally, make the funds spendable under any condition that can be expressed in Bitcoin Script.

We denote L_r^P as the output of invoking `read()` on party P at the end of round r . We restrict honest parties to only include *valid* transactions in their ledgers⁷. As we are interested in building BitVM on Bitcoin, when we present the construction, transactions are deemed (in)valid based on Bitcoin’s validation rules (see Section 3.1). However, BitVM can be built on top of any distributed ledger protocol with validation rules as expressive as those of Bitcoin. We assume that our protocol participants have access to the functionality exposed by the distributed ledger protocol, either by being an active participant or by running some (light) client protocol. We are interested in distributed ledger protocols that are safe and live, as defined below (cf. [25, 39, 7]). Given two sequences A and B , we use $A \preceq B$ to mean that A is a prefix of B .

► **Definition 2 (Safety).** *A distributed ledger protocol is safe, if (i) for any honest party P and any rounds $r_1 \leq r_2$, it holds that $L_{r_1}^P \preceq L_{r_2}^P$ and (ii) for any pair of honest parties P_1, P_2 and any pair of rounds r_1, r_2 , it holds that $L_{r_1}^{P_1} \preceq L_{r_2}^{P_2} \vee L_{r_2}^{P_2} \preceq L_{r_1}^{P_1}$.*

► **Definition 3 (Liveness).** *A distributed ledger protocol execution is $\text{live}(u)$, if any transaction that is written to an honest party’s ledger at round r , appears in the ledger of all honest parties by round $r + u$, denoted as L_{r+u} .*

Throughout this paper, we say “publish a transaction tx (on L)” to denote calling the function `write(tx)`. Furthermore, after publishing a valid transaction tx , we sometimes say “wait until tx appears (on L)”, to denote calling the function `read()` every round until $tx \in L$, which happens at most after u rounds due to liveness. When presenting the BitVM construction, we sometimes refer to the ledger as blockchain even though the distributed ledger protocol could be realized differently. We say something happens *on-chain* if there are one or more corresponding transactions in the ledger, and something happens *off-chain* if there are no corresponding transactions on the ledger.

There is a ledger state that is induced by a ledger L , denoted as $\text{st}(L)$, by executing each transaction in order, starting with a genesis state. The execution of transactions is captured by a state transition function, taking a state and a transaction and outputting a new state. We denote $\text{bal}_L(P) \in \mathbb{R}_{\geq 0}$ as the balance of party P in the state induced by L . A party can use parts of their balance $\text{in}_P \in [0, \text{bal}_L(P)]$ as *monetary input* for a transaction. For a given ledger L , we define the on-chain (monetary) utility of a transaction $tx \in L$ for a party P as $w_L(P, tx) := \text{bal}_{L_2}(P) - \text{bal}_{L_1}(P)$, where $L_1 \prec L$ is the ledger up to (not including) tx and $L_2 := L_1 || tx$. Usually, it is obvious which ledger we refer to, so we omit the subscript. In addition to balances of parties, a ledger state $\text{st}(L)$ can include a string $s \in \{0, 1\}^*$, denoted as $s \in \text{st}(L)$, if there exists a transaction $tx \in L$, such that tx contains the string s .

2.2 Threat model

We analyze BitVM in the presence of a PPT adversary that may corrupt any protocol party $\{P, V\}$ during the execution of the protocol. The adversary can corrupt parties, causing them to behave either as *Byzantine* or as *rational* actors. Byzantine parties can deviate arbitrarily from the honest protocol execution. Contrarily, rational parties deviate from the honest protocol execution only when such action increases their monetary utility.

⁷ This is not strictly necessary and is done mainly for convenience. Parties could also take an outputted ledger and remove invalid transactions from it.

The protocol gives different guarantees based on the type of corruption. On a high level, we want to show that (i) honest protocol participants are guaranteed their rightful balance even if the other party is Byzantine, (ii) rational parties follow the honest protocol execution, and (iii) if both parties behave rationally, the protocol follows an optimistic execution (which is efficient). We formally define these properties in Section 2.3.

2.3 Protocol goals

The core objectives of BitVM are termed *balance security* and *rational correctness*. Informally, balance security ensures an honest party will not lose their funds against Byzantine counterparties, whereas rational correctness guarantees that rational parties will follow the protocol. To formally define balance security we argue in terms of utility, i.e., the utility of the on-chain state of an honest party after the settlement of a BitVM instance will be at least equal to its utility of the correct final state, regardless of the actions of its counterparty. Rational correctness implies that if both parties are rational, they will commit on-chain the correct final state of the BitVM instance. These properties are standard in the literature: for instance, an honest user of a Lightning channel [36] can always dispute a malicious commitment and claim the channel funds, while rational players will always commit to the last agreed-upon state [37].

We formalize these objectives on a generic primitive, which we call *on-chain state verification* protocol and is defined as follows.

► **Definition 4** (On-chain State Verification Protocol). *An on-chain state verification protocol, parameterized over a distributed ledger protocol that outputs a ledger $\mathcal{L}$, is a two-party protocol that exposes the two following functionalities:*

- *setup($\text{in}_P, \text{in}_V, \Pi, f$): takes as input monetary inputs $\text{in}_P \in [0, \text{bal}_{\mathcal{L}}(P)]$ and $\text{in}_V \in [0, \text{bal}_{\mathcal{L}}(V)]$ of parties P and V , a computable function (or program) $\Pi : \mathcal{S} \rightarrow \mathcal{O}$ that maps a set of states $\mathcal{S}$ to a set of outcomes $\mathcal{O}$ and an outcome mapping function $f : \mathcal{O} \rightarrow \mathbb{R}_{\geq 0}^2$, that maps the set of outcomes $\mathcal{O}$ to pairs of utilities (v_P, v_V) where $v_P + v_V \leq \text{in}_P + \text{in}_V$ ⁸ and returns an instance $\mathcal{I}$.*
- *execute($\mathcal{I}, x$): takes as input an instance $\mathcal{I}$ returned by the setup function and a function input $x \in \mathcal{S}$ (for function Π).*

Consider an execution of this primitive for given inputs $\text{in}_P, \text{in}_V, \Pi, f$, where $\mathcal{I} \leftarrow \text{setup}(\text{in}_P, \text{in}_V, \Pi, f)$, and then $\text{execute}(\mathcal{I}, x)$ are called, and finish in round r . Let $\mathcal{T}$ be the set of transactions that are included in $\mathcal{L}_{r+u}^{\cap}$ as a result of this execution. Moreover, we denote the utility of party $A \in \{P, V\}$ in $f(\Pi(x))$ by $f_A(\Pi(x))$.

Balance Security. An execution achieves balance security, if it holds that $\sum_{tx \in \mathcal{T}} (w(A, tx)) \geq v_A$ where $v_A = f_A(\Pi(x))$, for any honest $A \in \{P, V\}$.

Rational Correctness. An execution achieves rational correctness, if P and V are rational and $\sum_{tx \in \mathcal{T}} (w(A, tx)) = v_A$ where $v_A = f_A(\Pi(x))$, for any $A \in \{P, V\}$ and $\Pi(x) \in \text{st}(\mathcal{L}_{r+u}^{\cap})$.

An on-chain state verification protocol achieves balance security and rational correctness, respectively, if for any $\text{in}_P, \text{in}_V, \Pi, f$ the probability that the corresponding execution does not achieve balance security and rational correctness, respectively, is negligible in κ .

⁸ We use an inequality here to account for potential fees paid during protocol execution.

3 Preliminaries

In this section, we present the necessary background concerning Bitcoin Script and the key primitives our construction builds upon.

Notation. Given a sequence $A := (a_1, \dots, a_n)$, $A[i]$ represents its i -th element. We use $A[i : j]$ to denote the subsequence $(a_i, \dots, a_j)$. We use $|A|$ to denote the length of a sequence, e.g., $|(a_1, \dots, a_n)| = n$. For a string $s \in \{0, 1\}^*$, we use $|s|_{bit}$ to denote its bit length. For the full notation used throughout the paper, we refer the reader to the full version of the paper [5, Appendix A].

3.1 Transactions in the UTXO model

A user U on a ledger L is identified by the secret-public key pair (pk_U, sk_U) ; by $\sigma_U(m)$ we denote the digital signature of U over the message $m \in \{0, 1\}^*$.

In the *unspent transaction output* (UTXO) model, a transaction Tx maps a (non-empty) list of existing, unspent, transaction outputs to a (non-empty) list of new transaction outputs. A transaction output is defined as an attribute tuple $out := (a\text{B}, lockScript)$, where $out.a \in \mathbb{R}_{\geq 0}$ is the amount of coins (expressed in B) held by the output out and $out.lockScript$ is the condition that needs to be fulfilled to spend it and transfer the coins to a new output, which we also call UTXO. We distinguish the already existing transaction outputs (input of a transaction Tx) from the newly created outputs, calling them $Tx.inputs$ and $Tx.outputs$, respectively. A transaction input in is defined as $in := (PrevTx, outIndex, lockScript)$, where the output being spent is uniquely identified by specifying the transaction $PrevTx$ and an output index $outIndex$. To improve readability, we also give the locking script $lockScript$ that is being fulfilled and we specify the different transaction spending conditions.

In this paper, we use the following subset of Bitcoin spending conditions (which we describe in the full version of the paper [5, Appendix B]): signature locks, multisignature locks, relative timelocks, and taproot trees. We use these conditions as building blocks, combining them with standard Bitcoin Script constructions using logical operators $\wedge$ (and) and $\vee$ (or).

We formally define a transaction as a tuple $Tx := (inputs, witnesses, outputs)$ where $Tx.inputs := [in_1, \dots, in_n]$ are the transaction inputs, $Tx.outputs := [out_1, \dots, out_m]$ are the transaction outputs and $Tx.witnesses := [w_1, \dots, w_n]$ represents the witness data, i.e., the list of the tuples that fulfill the spending conditions of the inputs, one witness for each input. The locking script of an output is expressed in the scripting language of the ledger. To transfer the coins held in a UTXO, its locking script is executed with a witness as script input and must return `True`; if successful, the condition is considered fulfilled. If the script execution returns `False`, the condition is not fulfilled and the UTXO is not spendable⁹.

A transaction is valid only if every UTXO in input is unspent, the witnesses fulfill the conditions of the corresponding locking scripts, and the sum of the coins held in the inputs is equal to or greater than the sum of the coins held in the outputs.

⁹ In this work, we separate the locking script from the witness for readability. In practice, the protocol is implemented using SegWit [32] transactions, where the locking script is included in the witness.

3.2 Lamport digital signature scheme

Let $h : X \rightarrow Y$ be a one-way function, where $X := \{0, 1\}^*$ and $Y := \{0, 1\}^\lambda$, for a given security parameter λ . Let $m \in \{0, 1\}^\ell$ be a ℓ -bit message, with $\ell \in \mathbb{N}_{>0}$. A *Lamport digital signature scheme* [28] **Lamp** consists of a triple of algorithms (**KeyGen**, **Sig**, **Vrfy**), where:

- $(pk_{\mathcal{M}}, sk_{\mathcal{M}}) \leftarrow \text{Lamp.KeyGen}(\ell)$ (cf. Algorithm 1), is a Probabilistic Polynomial Time (PPT) algorithm that takes as input a positive integer ℓ and returns a key pair, consisting of a secret key $sk_{\mathcal{M}}$ and a public key $pk_{\mathcal{M}}$ which can be used for one-time signing an ℓ -bit message. We use $\mathcal{M} = \{0, 1\}^\ell$ as an alias for the ℓ -bit message space.
- $c_m \leftarrow \text{Lamp.Sig}_{sk_{\mathcal{M}}}(m)$ (cf. Algorithm 2), is a Deterministic Polynomial Time (DPT) algorithm parameterized by a secret key $sk_{\mathcal{M}}$, that takes as input a message $m \in \mathcal{M}$ and outputs the signature c_m , which we also call (Lamport) commitment.
- $\{\text{True}, \text{False}\} \leftarrow \text{Lamp.Vrfy}_{pk_{\mathcal{M}}}(m, c_m)$ (cf. Algorithm 3), is a DPT algorithm parameterized by a public key $pk_{\mathcal{M}}$ that takes as input a message m , a signature c_m , and outputs **True** iff c_m is a valid signature for m generated by the secret key $sk_{\mathcal{M}}$, corresponding to $pk_{\mathcal{M}}$, i.e., $(pk_{\mathcal{M}}, sk_{\mathcal{M}})$ is a key pair generated by **Lamp.KeyGen**.

■ **Algorithm 1** The key generation algorithm **Lamp.KeyGen** for a ℓ -bit message space $\mathcal{M}$. In the following algorithms, we use matrix notation, i.e., for a given two-dimensional matrix a , $a[i, j]$ refers to the element at row i and column j of it.

```

1: function Lamp.KeyGen( $\ell$ )
2:   Let  $sk_{\mathcal{M}} \leftarrow \begin{pmatrix} x[0, 0], \dots, x[0, \ell - 1] \\ x[1, 0], \dots, x[1, \ell - 1] \end{pmatrix}$ , where every element  $x[i, j]$  is sampled uniformly at
   random from the set  $X$ ;
3:   for  $i = 0, 1$  and  $j = 0, \dots, \ell - 1$  do
4:      $y[i, j] \leftarrow h(x[i, j])$ ;
5:   Let  $pk_{\mathcal{M}} \leftarrow \begin{pmatrix} y[0, 0], \dots, y[0, \ell - 1] \\ y[1, 0], \dots, y[1, \ell - 1] \end{pmatrix}$ ;
6:   return  $(sk_{\mathcal{M}}, pk_{\mathcal{M}})$ .
```

■ **Algorithm 2** The Lamport signature algorithm **Lamp.Sig**, parameterized over a secret key $sk_{\mathcal{M}}$ for a ℓ -bit sized message space $\mathcal{M}$.

```

1: function LampSig $_{sk_{\mathcal{M}}}(m)$ 
2:   for  $i = 0, \dots, \ell - 1$  do
3:     Let  $c_m[i] \leftarrow sk_{\mathcal{M}}[m[i], i]$ ;
4:   return  $c_m$ .
```

■ **Algorithm 3** Lamport verification algorithm **Lamp.Vrfy**, parameterized over a public key $pk_{\mathcal{M}}$ for a ℓ -bit message space $\mathcal{M}$.

```

1: function Lamp.Vrfy $_{pk_{\mathcal{M}}}(m, c_m)$ 
2:   for  $i = 0, \dots, \ell - 1$  do
3:     if  $h(c_m[i]) \neq pk_{\mathcal{M}}[m[i], i]$  then
4:       return False;
5:   return True.
```

Lamport signatures are secure one-time signatures. Given a message space $\mathcal{M}$, it is possible to sign any message $m \in \mathcal{M}$ by using the secret key $sk_{\mathcal{M}}$ of the key pair $(sk_{\mathcal{M}}, pk_{\mathcal{M}})$, i.e., the key pair associated to $\mathcal{M}$. When the message m is signed and c_m is created, the key

pair becomes bound to m . No polynomially bounded adversary is able to forge a signature for a different message $m' \neq m$ with non-negligible probability. However, if the signer uses the same secret key $sk_{\mathcal{M}}$ to sign another different ℓ -bit message $m'' \neq m$, they can be held accountable. We call this action *equivocation* and we show how to detect it in Algorithm 4.

Notice that signing the ℓ -bit message m with the secret key $sk_{\mathcal{M}}$ consists in revealing for every bit $i = 0, \dots, \ell - 1$ of m one of the two preimages that compose the i -th column of secret key $sk_{\mathcal{M}}$, namely, revealing $x[0, i]$ to claim that $m[i] = 0$, or revealing $x[1, i]$ to claim that $m[i] = 1$. When the signer reveals both $x[0, i], x[1, i]$ for any bit i , they are equivocating.

For a formal discussion about one-time security and a proof that Lamport signatures are one-time secure (assuming the existence of one-way functions), see [15]. One-time security is crucial for the correctness of BitVM as it enables the signer of a message to make a non-repudiable commitment to that message. Lamport signatures are implementable using Bitcoin Script, as demonstrated in [2].

■ **Algorithm 4** The CheckEquivocation algorithm for a bit $b \in \mathcal{B} = \{0, 1\}$. The input is the corresponding public key $pk_{\mathcal{B}}$ and two preimages $x', x'' \in X$.

```

1: function CheckEquivocation( $pk_{\mathcal{B}}, x', x''$ )
2:   if ( $h(x') = pk_{\mathcal{B}}[0, 0]$  and  $h(x'') = pk_{\mathcal{B}}[1, 0]$ ) then
3:     return True;
4:     ▷ The committer is trying to commit to both 0 and 1 for the bit  $b$ .
5:   else
6:     return False.
```

In the following, we are interested in Lamport signatures as a mechanism to enable a party to *commit* to (single or multiple bits) messages. Thus, we will refer to Algorithm 2 as Comm instead of Lamp.Sig and to Algorithm 3 as CheckComm instead of Lamp.Vrfy.

3.3 Stateful Bitcoin scripting

Although Bitcoin's scripting language is stateless, clever use of one-time digital signature schemes (e.g., Lamport signatures) enables state preservation across Bitcoin transactions.

Consider the following example: Let a user U hold a Lamport key pair $(sk_{\mathcal{M}}, pk_{\mathcal{M}})$ associated with $\mathcal{M}$, the set of all ℓ -bit messages. We can think of $\mathcal{M}$ as a variable that can hold any ℓ -bit string. U can assign a value m to $\mathcal{M}$ by creating the commitment $c_m \leftarrow \text{Comm}_{sk_{\mathcal{M}}}(m)$.

By hard-coding $\text{CheckComm}_{pk_{\mathcal{M}}}$ for a public key $pk_{\mathcal{M}}$ in the locking script of multiple outputs, this variable assignment can not only be verified but also transferred from one output to another, effectively establishing a global state in Bitcoin. This is accomplished by reading m and c_m from the unlocking script of one output and passing them to another output through its witness. For example, consider two different transactions $\text{Tx}_1 := (*, *, [\text{out}_1, *])$ and $\text{Tx}_2 := (*, *, [\text{out}'_1, *])$, where the outputs are defined as $\text{out}_1 := (a\$, \text{CheckComm}_{pk_{\mathcal{M}}})$ and $\text{out}'_1 := (b\$, \text{CheckComm}_{pk_{\mathcal{M}}})$. To unlock both out_1 and out'_1 , a Lamport commitment c_m must be provided. Since the same Lamport public key appears in both scripts, every party in the network knows that when U unlocks these scripts, U is assigning a value to the same variable $\mathcal{M}$. Following from *one-time security*, no user other than U can assign a different value to $\mathcal{M}$ without knowing $sk_{\mathcal{M}}$. Moreover, U cannot assign two different values $m_1 \neq m_2$ to $\mathcal{M}$ without equivocating, which is detectable and can be punished on-chain.

4 BitVM Virtual Machine

In the BitVM protocol, both parties employ a *Virtual Machine* (VM) to run off-chain any deterministic program Π . Although the underlying concept closely resembles an *abstract machine*, we choose to retain the term “VM” to stay consistent with the original naming of the construction. In this section, we describe the components of the VM and demonstrate how to initialize them for practical deployment of the protocol.

VM components. At a high level, the virtual machine (VM) executes programs composed of instructions written in a VM-compatible language. While the program is running, the VM continuously performs an instruction cycle, or *state transition function*. In each cycle, the VM fetches the instruction indicated by the program counter, loads the values stored at specific memory addresses referenced by the instruction, executes the operation defined by the instruction on those values, stores the result at the designated memory address, and updates the program counter accordingly (cf. Definition 6).

This process repeats until the program terminates or reaches a predefined execution limit. Throughout its execution, the VM produces an execution trace, recording (i) the current program counter value and (ii) a commitment to the state of memory at each step. The BitVM protocol leverages this execution trace for dispute resolution, as described in the full version of the paper [5, Appendix C.3, C.4].

Formally, let a *VM address* be an integer $addr \in \mathcal{A} := \{0, 1, \dots, \text{MemLen} - 1\}$ where $\text{MemLen} \in \mathbb{N}_{>0}$ represents the memory length. We define the VM *memory* as the sequence $M \in \mathcal{M} := \{0, 1, \dots, n\}^{\text{MemLen}}$, where $n \in \mathbb{N}_{>0}$ specifies the range of values stored at any memory address. The *VM program counter*, denoted pc , is an element of the set $\mathcal{PC} := \{0, 1, \dots, \ell - 1\} \cup \{\perp\}$, where $\ell \in \{1, 2, \dots, n\}$ is the maximum length of the program, and $\perp$ indicates termination. Let $\mathcal{OP} := \{f_{OP} : \mathcal{PC} \times \{0, \dots, n\} \times \{0, \dots, n\} \rightarrow \mathcal{PC} \times \{0, \dots, n\} \cup \{\perp\}\}$ be a set of CPU instructions that the VM can execute¹⁰. The function f_{OP} takes as input a triple (pc, val_A, val_B) and outputs a pair (pc, val_C) . For any CPU instruction $f_{OP} \in \mathcal{OP}$, we require that f_{OP} is executable in Bitcoin Script. A *VM program* is an ordered sequence of ℓ elements, denoted $\Pi \in \mathcal{I}^\ell$, where $\mathcal{I} := \{(f_{OP}, addr_A, addr_B, addr_C) \mid addr_A, addr_B, addr_C \in \mathcal{A}, f_{OP} \in \mathcal{OP}\}$. We can now define the following.

► **Definition 5 (VM State).** A VM state, or simply, state, is a triple $S := (M, pc, \Pi)$, where M is the VM memory, pc is the VM program counter, and Π is a VM program.

► **Definition 6 (State Transition Function).** Let $\mathcal{S} := \mathcal{M} \times \mathcal{PC} \times \mathcal{I}^\ell$ be the set of all VM states. We define the state transition function $f_{ST} : \mathcal{S} \rightarrow \mathcal{S}$ with f_{ST} taking as argument the state (M_i, pc_i, Π) and giving as output the state (M_{i+1}, pc_{i+1}, Π) as specified in Algorithm 5.

Given a program Π and a memory configuration M , we assume that the entry point of the program, namely the first instruction that a program executes, is always $\Pi[0]$. Thus, we define as *initial* state the tuple $S_0 := (M, 0, \Pi)$. We use the shorthand notation $f_{ST}^i(S)$ when we apply the state transition function f_{ST} to a state S exactly i times, $f_{ST}^i(S) := f_{ST}(f_{ST}(\dots(f_{ST}(S))))$. We say that a state $S_i := (M_i, pc_i, \Pi)$ at step i is *correct* with respect to an initial state S_0 iff $S_i = f_{ST}^i(S_0)$. We avoid the subscripts (and refer to S_i as (M, pc, Π)) when it is clear from the context which state we are referring to. Finally, after a number of execution steps equal to *final* (*final* is decided when a VM instance is created), the program terminates. We denote the final state, or outcome, as $\Pi(S_0) := f_{ST}^{\text{final}}(S_0)$.

¹⁰ Even though $\mathcal{OP}$ can be arbitrary, we are interested in a Turing-complete instruction set. In particular, we later use ADD, BEQ, and JMP, cf. Algorithm 7 – a well-known Turing-complete instruction set [21].

■ **Algorithm 5** State Transition Function f_{ST} .

```

1: function  $f_{ST}(M, pc, \Pi)$ 
2:   if  $pc = \perp$  then
3:     return  $(M, pc, \Pi)$ .
4:    $M' \leftarrow M$ ;
5:    $(f_{OP}, addr_A, addr_B, addr_C) \leftarrow \Pi[pc]$ ;
6:    $val_A \leftarrow M[addr_A]$ ;
7:    $val_B \leftarrow M[addr_B]$ ;
8:    $(pc', val_C) \leftarrow f_{OP}(pc, val_A, val_B)$ ;
9:   if  $val_C \neq \perp$  then
10:     $M'[addr_C] \leftarrow val_C$ ;
11:  return  $(M', pc', \Pi)$ .

```

We define a VM instance as a tuple $\Gamma := \langle \Pi, \text{MemLen}, n, \text{final} \rangle$. We write Γ^A to refer to the VM instance executed by party A . We write S_i^A to denote a VM state S_i that A claims to have produced during the execution of A 's VM instance Γ^A . We say that two parties A and B agree on the state S_i if $S_i^A = S_i^B$, and disagree on S_i otherwise.

► **Definition 7** (Execution Trace Element). *Let $(M_i, pc_i, \Pi) := f_{ST}^i(S_0)$, and let MR_i be the root of the Merkle tree with the entries of M_i as its leaves. The i -th VM execution trace element, or simply, i -th trace element is the pair $E_i := (MR_i, pc_i)$, for $i \in \{0, \dots, \text{final}\}$.*

We write E_i^A to denote a VM execution trace element E_i that A claims to have produced during the execution of A 's VM instance Γ^A . The VM *execution trace* is defined as a sequence of consecutive trace elements $ExecTrace := (E_0, \dots, E_{\text{final}})$. We write $ExecTrace^A$ as a short-hand for $(E_0^A, \dots, E_{\text{final}}^A)$.

We describe how the VM behaves in Algorithm 6: starting from initial state S_0 , it applies the state transition function f_{ST} to the state and records the related trace elements until the program Π ends, namely, once pc is set to be $\perp$. The VM algorithm is parameterized by final , a parameter that represents the maximum number of state transitions that the VM is allowed to perform. The VM algorithm returns as output the VM execution trace $ExecTrace$, along with the resulting memory M after the program execution.

■ **Algorithm 6** The VM algorithm. S_0 is the initial VM state.

```

1: function  $\text{VM}_{\text{final}}(S_0)$ 
2:    $stepCount \leftarrow 0$ ;
3:   while  $stepCount < \text{final}$  do
4:      $E_{stepCount} \leftarrow (MR, pc)$ ;
5:      $(M, pc, \Pi) \leftarrow f_{ST}(M, pc, \Pi)$ ;
6:     increment  $stepCount$  by 1;
7:    $E_{stepCount} \leftarrow (MR, pc)$ ;
8:    $ExecTrace \leftarrow (E_0, \dots, E_{\text{final}})$ ;
9:  return  $(ExecTrace, M)$ .

```

A practical VM instance. For better readability and to provide a protocol instance that can be deployed in practice, in the rest of the paper, we will consider a VM instance $\Gamma := \langle \Pi, \text{MemLen}, n, \text{final} \rangle$ with the following initialization: We set the length of the memory

as $\text{MemLen} = 2^{32}$ and the greatest integer that can be stored in any entry of the memory as $n = 2^{32} - 1$. Furthermore, we assume that the input program Π has $\ell \leq 2^{32}$ number of instructions¹¹ and we set $\text{final} = 2^{32}$.

As for the set $\mathcal{OP}$ of instructions that the VM can execute, our VM instance employs the following: $\mathcal{OP} := \{\text{ADD}, \text{BEQ}, \text{JMP}\}$. This is a minimal set of computer instructions known to be Turing complete [21]. We underscore that the BitVM protocol can function with any Turing-complete instruction set, provided that each instruction within the set is implementable in Bitcoin script. In Algorithm 7, we give an implementation of ADD, BEQ and JMP that can be easily translated in Bitcoin script.

■ **Algorithm 7** The Algorithms ADD, BEQ, and JMP, each taking as input the tuple (pc, val_A, val_B) , and returning a pair (pc, val_C) .

```

1: function ADD( $pc, val_A, val_B$ )
2:   if  $pc = \perp$  then return  $(\perp, \perp)$ ;
3:   return  $(pc + 1, val_A + val_B)$ .

4: function BEQ( $pc, val_A, val_B$ )
5:   if  $pc = \perp$  then return  $(\perp, \perp)$ ;
6:   if  $val_A = val_B$  then return  $(pc + 1, \perp)$ ;
7:   else return  $(pc + 2, \perp)$ ;

8: function JMP( $pc, val_A, val_B$ )
9:   if  $pc = \perp$  then return  $(\perp, \perp)$ ;
10:  return  $(val_A, \perp)$ .

```

5 The BitVM Protocol

The BitVM protocol enhances Bitcoin’s expressiveness by enabling spending conditions based on the result of general-purpose computation – performed off-chain, but verifiable on-chain through an interactive protocol. While Bitcoin Script is not (quasi-)Turing complete, BitVM effectively simulates such computation by using cryptographic commitments, economic incentives, and Script-compatible fraud proofs.

BitVM enables two mutually distrusting parties, the prover (P) and the verifier (V), to agree on the output of a program Π run on input S_0 . If both parties agree on the outcome, funds are distributed accordingly with minimal on-chain interaction. In case of disagreement, BitVM executes a dispute resolution protocol that isolates the exact point of divergence and verifies correctness using Bitcoin Script. All necessary transaction logic is expressible within Bitcoin, utilizing Lamport signatures and other Bitcoin Script features, thus no consensus changes are needed.

In the following, we provide a high-level description of the protocol’s logic, and conclude with a discussion on its security guarantees and on-chain performance. The full protocol specification and concrete transaction constructions are deferred to the full version of the paper [5, Appendix C].

¹¹In the BitVM protocol, we build a Taproot tree where every program instruction is a Tapleaf script. We chose such ℓ since $2^{32} \ll 2^{128}$, the maximum number of leaf scripts in the current specification of Bitcoin [41].

Protocol Phases. The BitVM protocol proceeds in four phases:

1. **Setup:** P and V agree on:
 - A program Π written for the BitVM virtual machine.
 - An outcome mapping function f , defining fund redistribution based on program output.
 - A time parameter Δ , used to timelock transactions to deter inactivity,
 - and a maximum execution trace length $\text{final} = 2^{32}$.
 They presign all necessary transactions, including outcomes and disputes, and publish a **Setup** transaction locking funds on-chain in a multisig with timeout clauses.
2. **Execution:** P communicates the initial input S_0 to V . Then, both parties compute $\Pi(S_0)$ off-chain, producing an execution trace $\text{ExecTrace} = (E_0, \dots, E_{\text{final}})$, where each E_i is an execution trace element as in Definition 7.
3. **Commitment:** P posts **CommitComputation**, committing to both S_0 and the claimed output $\Pi(S_0)$ using Lamport-based commitments. If V agrees, P finalizes with **Close**, spending funds according to f . Otherwise, a dispute begins.
4. **Dispute Resolution:** If V disagrees with the claimed output, V engages P in an on-chain bisection game over the execution trace to identify a disputed VM step. Once found, this VM step is executed on-chain using Bitcoin Script. If the result is different from what P claimed, their funds are forfeited.

Optimistic Case (Happy Path)

If both parties agree on the result of the off-chain computation, only three transactions are posted on-chain:

1. **Setup** – locks funds in a multisig.
2. **CommitComputation** – P commits to the input and the output of the program using Lamport signatures.
3. **Close** – P spends **CommitComputation**’s transaction output with a result consistent with f , redistributing the funds.

Dispute Resolution (Unhappy Path)

If V disagrees with P ’s committed result, the protocol enters the dispute phase. Let $\text{ExecTrace} := (E_0, \dots, E_{\text{final}})$ be the off-chain VM execution trace, where each element is defined as in Definition 7. Notably, each successive element in this execution trace results from applying a single VM instruction to the preceding element. Since P and V agree on S_0 ¹², any disagreement on the result implies a disagreement on some S_{i+1} , while agreeing on S_i . The bisection game locates such an index $\mathcal{N}$, enabling on-chain verification of a single VM step (we formalize the bisection game in the full version of the paper [5, Appendix D]). Figure 1 illustrates an overview of the dispute process.

Identify Disagreement. V initiates the dispute by publishing the **KickOff** transaction, starting a 32-round challenge-response game. In each round j , V commits to a bit b_{32-j} via **TraceChallenge_j**, directing the search left ($b_{32-j} = 0$) or right ($b_{32-j} = 1$).

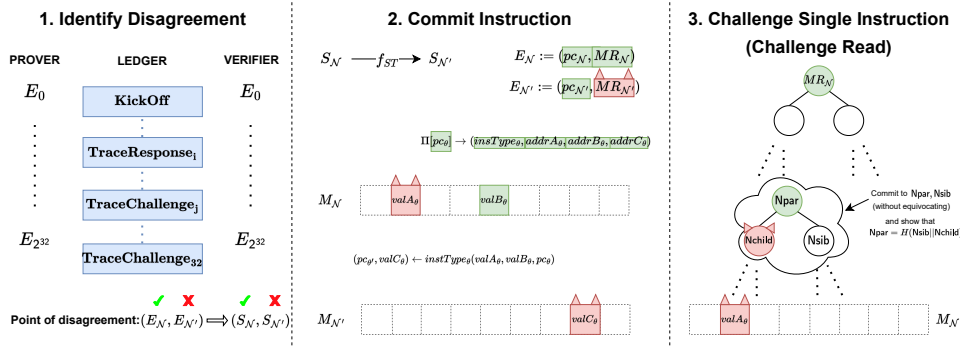
P responds with **TraceResponse_{j+1}**, revealing the midpoint E_{n_r} , where $r = 32 - (j + 1)$ and $n_r = \sum_{i=r}^{31} b_i \cdot 2^i + 2^r$. After 32 rounds, the final index is $\mathcal{N} = \sum_{i=0}^{31} b_i \cdot 2^i$, with $\mathcal{N}' = \mathcal{N} + 1$. This yields the first divergent pair: agreement on $S_{\mathcal{N}}$, disagreement on $S_{\mathcal{N}'}$.

¹²They agree by default, since P chooses the initial state S_0 and communicates it to V .

Instruction Commitment. P must now prove that $S_{N'} = f_{ST}(S_N)$. To do so, they publish a **CommitInstruction** transaction, committing to:

- $pc_\theta = pc_N$ and $pc_{\theta'} = pc_{N'}$: the program counters of the states S_N and $S_{N'}$, respectively;
- $insType_\theta \in \mathcal{OP} := \{\text{ADD}, \text{BEQ}, \text{JMP}\}$: the instruction type at $\Pi[pc_\theta]$;
- $addrA_\theta, addrB_\theta, addrC_\theta$: the memory addresses referenced in $\Pi[pc_\theta]$;
- $valA_\theta, valB_\theta$: the memory values at addresses $addrA_\theta, addrB_\theta$ in S_N ;
- $valC_\theta$: the memory value at address $addrC_\theta$ in $S_{N'}$, i.e., after executing $f_{ST}(S_N)$.

The Taproot script must enforce that: $(pc_{\theta'}, valC_\theta) = insType_\theta(pc_\theta, valA_\theta, valB_\theta)$. Next, V may challenge this step via several failure cases.



■ **Figure 1** Example of dispute resolution in the BitVM protocol: To resolve a dispute, (1) P and V engage in a bisection game to identify the point of disagreement $(E_N, E_{N'})$ in their execution traces, indicating a disagreement in the transition from state S_N to $S_{N'}$. Next, (2) P commits on-chain to all necessary information for executing $S_{N'} = f_{ST}(S_N)$ (i.e., the values highlighted by colored boxes in the figure). In this example, we assume that P is committing to an incorrect value for $valA_\theta$, resulting in incorrect value for $valC_\theta$ and $MR_{N'}$. (3) V challenges P through a bisection game over the path in the memory Merkle tree M_N , from the root MR_N to the leaf containing $valA_\theta$. This bisection game reveals two intermediate nodes, $Npar$ and $Nchild$, on which P and V disagree. To get away while using an incorrect value, P would need to prove that $Nchild$ is indeed the left child of $Npar$, which is impossible without equivocating, leading to punishment.

Verifier Challenges. There are five challenge paths to verify the correctness of the transition $S_N \rightarrow S_{N'}$. In essence, either the values are not correctly derived from the agreed upon state S_N , or the result is not stored correctly in $S_{N'}$. By executing this step, V can pinpoint which of these errors occurred, and thus choose the appropriate challenge path.

(i)-(ii) Program Counter (Current/Next). V can dispute that $pc_\theta \neq pc_N$ or $pc_{\theta'} \neq pc_{N'}$ via **ChallengeCurrPC** or **ChallengeNextPC**. If proven, V claims the funds. If V equivocates on their bitwise commitment to N , P can reveal equivocation via **PunishCurrPC**, claiming all funds instead.

(iii) Instruction Mismatch. Each program line (indexed by pc_θ) is encoded in a tapleaf with the expected $insType_\theta, addrA_\theta, addrB_\theta$, and $addrC_\theta$. If P commits to inconsistent values, V can reveal this via **DisproveProgram**, spending the corresponding tapleaf. If the program counter matches but any operand or opcode does not, V wins the dispute and claims the funds.

(iv) Read Error. To verify $valA_\theta = M_N[addrA_\theta]$, V initiates a challenge by publishing **ChallengeRead**. A 5-round bisection game over the Merkle path $\mathcal{P}_R := (MR_N, \dots, M_N[addrA_\theta])$, i.e. the path from the root MR_N to the leaf $M_N[addrA_\theta]$, isolates a disagreement $(Npar, Nchild)$. If P fails to justify it via a sibling hash $Nsib$ (such that $H(Nsib || Nchild) = Npar$) or equivocates on one of the elements of $\mathcal{P}_R$, V wins via **PunishRead**. Otherwise, P claims the funds after a timeout. The same logic applies to $valB_\theta$ at address $addrB_\theta$.

(v) Write Error. V challenges $valC_\theta \neq M_N[addrC_\theta]$ by publishing **ChallengeWrite**. The bisection game now runs on two Merkle paths (before and after the write). Again, V identifies a disagreement $(Npar, Nchild)$ and challenges P to provide a consistent sibling node. If P cannot do so, V wins via **PunishWrite**; otherwise, P claims the funds after a timeout.

6 Security Analysis

We now show that BitVM is an *on-chain state verification protocol* that satisfies two key properties: *Balance Security*, which ensures that honest users never lose funds even if their counterparty behaves arbitrarily; and *Rational Correctness*, which guarantees that rational participants always follow the intended optimistic execution path. Our analysis models BitVM as an *Extensive Form Game (EFG)* (we define it in the full version of the paper [5, Appendix E]), which enables unified reasoning about both Byzantine and rational adversaries. This approach builds on recent work on incentive-compatible Layer-2 protocols [37], and provides a natural way to establish equilibrium guarantees, which are essential in financial settings.¹³

► **Theorem 8.** BitVM is an on-chain state verification protocol that achieves balance security and rational correctness.

Balance Security

We consider two cases: (i) both parties behave honestly, and (ii) one party $A \in \{P, V\}$ deviates at any step. In both scenarios, we prove that the honest party does not lose their funds. We note that if either party deviates during *Setup*, the honest party will refuse to sign the **Setup** transaction, ensuring no coins are locked unless both parties have received all necessary pre-signed transactions ([5, Lemma F.1]). Thus, we assume that the setup phase has concluded successfully.

Honest parties. When both parties are honest, BitVM follows an optimistic path: the prover posts the correct computation result on-chain via **CommitComputation**, and after the timelock expires, publishes **Close** to distribute the funds according to f ([5, Lemma F.3]).

¹³While Universal Composability (UC) is well-suited for modeling arbitrary adversaries, it does not support equilibrium reasoning. Formal tools targeting rational security exist for simple constructions (e.g., Lightning’s closing game [16]), but do not scale to BitVM’s combinatorial structure (e.g., bisection-based disputes). Developing general-purpose frameworks for rational security in expressive smart contract systems remains an open challenge.

V honest, P Byzantine. If the prover fails to publish in time `CommitComputation`, either due to inactivity or incorrect computation, or subsequently fails to post `Close`, the verifier can reclaim all the funds locked in the multisignature after the respective timelocks expire ([5, Lemma F.2, F.3]). This mechanism prevents hostage scenarios by ensuring that the verifier can recover their coins in case of non-responsiveness.

If the prover commits to an incorrect result in `CommitComputation`, the verifier initiates the Identify Disagreement phase by publishing `KickOff`. If the prover remains inactive, the verifier can claim the coins after the timelock expires ([5, Lemma F.4]). If the phase completes, the verifier obtains a VM step for which the prover has incorrectly committed to the outcome of the state transition function (Algorithm 5) ([5, Lemma F.7]).

The prover may have deviated by using an invalid program counter (current or next), performing an incorrect memory read or write, or executing an invalid instruction. For each case, the verifier can post the corresponding on-chain transaction – such as `ChallengeCurrPC`, `ChallengeNextPC`, `ChallengeRead`, `ChallengeWrite`, or `DisproveProgram` – to initiate the appropriate dispute path. This allows the verifier to disprove the prover’s computation and claim the funds ([5, Lemma F.14]).

P honest, V Byzantine. A malicious verifier may initiate the Dispute Phase by publishing `KickOff` on-chain, even though the prover has correctly committed to the result in `CommitComputation`. If the verifier becomes inactive during the Identify Disagreement phase, the prover can claim the funds once the timelock expires ([5, Lemma F.5]).

If the phase completes, the verifier must follow up with a challenge by posting one of the transactions `ChallengeCurrPC`, `ChallengeNextPC`, `ChallengeRead`, `ChallengeWrite`, or `DisproveProgram`. Since the prover’s commitment is correct, the verifier cannot produce a valid inconsistency and ultimately fails to disprove the computation. In this case, the prover reclaims the funds ([5, Lemma F.15]).

Rational Correctness

We now establish that rational participants are incentivized to follow the optimistic execution path of BitVM. Specifically, in [5, Theorem D.16], we prove that the honest strategy profile constitutes a *Subgame Perfect Nash Equilibrium (SPNE)*. The proof proceeds by backward induction on the game tree: in every subgame, deviation results in strictly lower utility, since the honest counterparty can either recover funds via timeouts or successfully disprove incorrect behavior on-chain. These consequences are established through the same mechanisms formalized in [5, Lemma F.2, F.15].

In particular, if the prover deviates by omitting `CommitComputation`, failing to post `Close`, or committing an invalid transition, the verifier can either reclaim their funds or win the dispute. Conversely, if the verifier initiates an unwarranted dispute, they will be unable to disprove the prover and ultimately forfeit their claim. As any unilateral deviation leads to lower utility, both parties are incentivized to behave honestly. This makes BitVM incentive-compatible: rational players follow the optimistic path without invoking a dispute.

7 Implementation and Evaluation

To show the feasibility of our approach, we implement a prototype of BitVM in JavaScript [2]. In addition to showing how BitVM can be implemented practically, we use it to compute the transaction fees for both an optimistic run and the most expensive dispute branch of BitVM.

We assume constant transaction fees of $3\text{sat}/vB^{14}$, a Bitcoin price of \$73,000 (as of May 2026). To make the prototype more efficient, we realize the one-time signatures with Winternitz signatures [34] instead of Lamport signatures. Using Winternitz signatures, both the size of a signature and the size of a public key are around $5vB$ per message bit. As hash function, we use the Bitcoin Script primitive `OP_HASH160` [35]. We also assume that $\Delta = 12$ hours, meaning each timelock expires after half a day. Different concrete values can be chosen, but any such selection would require scaling the time evaluation accordingly.

Optimistic case. In the optimistic case, three on-chain transactions are published: **Setup**, **CommitComputation**, and **Close**, totaling $1,944vB^{15}$. The protocol’s execution cost is $5,832$ sat (\$4.26). In terms of execution time, once **Setup** is published, BitVM completes in at most 2Δ time, corresponding to 1 day.

Dispute case. We focus on the most expensive path in terms of fees, the *Write Error* path. Overall, 81 transactions are posted on-chain; the path weighs $244,040vB$. The total protocol execution cost is 732ksat , or about 534\$, and, once the **Setup** transaction is published on-chain, it takes at most $80 \times \Delta = 40$ days to complete its execution. In case of a dispute, all the fees needed to run the protocol on-chain are covered by the misbehaving party¹⁶.

These costs are acceptable for a market deployment, as witnessed by the massive investments of the industry in the BitVM paradigm [23, 38, 17].

8 Bridge Application

In this section, we leverage BitVM to instantiate a bridge application between the Bitcoin ledger and a sidechain system running a distributed ledger protocol, as defined in Definition 1, which satisfies *safety* and *liveness*. This bridge enables users to mint (wrapped) Bitcoin tokens on the sidechain and later redeem them back on the Bitcoin blockchain and is secure, assuming only existential honesty of the participants.

We first outline the bridge protocol, then present some high-level security arguments and conclude this section with an evaluation.

8.1 A BitVM-based Bridge Design

Consider two users, Alice and Bob: Alice mints tokens on the sidechain, transfers them to Bob, who then redeems the equivalent amount back to Bitcoin. The protocol assumes a committee of n members active during the `BridgeSetup` phase¹⁷, with at least one assumed honest during execution. We illustrate the bridge protocol in Figure 2.

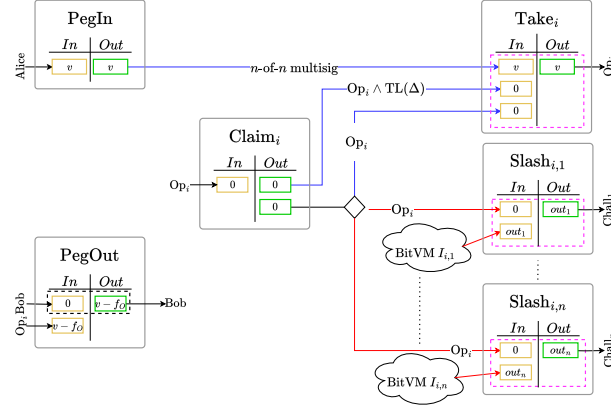
During the *BridgeSetup* phase, the committee members pre-sign the transactions required for the protocol execution. The core procedures are *PegIn* (Alice mints tokens) and *PegOut* (Bob redeems them). To enable **PegOut**, a committee member fronts coins to Bob and later

¹⁴In Bitcoin, the size of a SegWit [32] transaction is expressed in *virtual Bytes*, or *vBytes* (vB). The number of vB of a transaction witness is equal to its number of Bytes divided by four.

¹⁵Verifying an arbitrary program with BitVM is $5\times$ more expensive than a 2-input standard pay-to-public-key-hash transaction ($374vB$).

¹⁶Since we do not know beforehand if a party will misbehave, both P and V need to put enough collateral to cover for the fees. To keep low the opportunity cost of running BitVM, we aim to have a protocol with minimal on-chain footprint.

¹⁷Otherwise, the committee is reshuffled until `BridgeSetup` completes.



■ **Figure 2** Illustration of BitVM-based bridge. For readability, we represent transactions as grey boxes; the value carried by transaction inputs and outputs is written inside a yellow and green rectangle, respectively. Pink dashed rectangles around the inputs and outputs of **Take** and **Slash** denote that the transaction is pre-signed during BridgeSetup, while the black dashed rectangle indicates which portion of the PegOut transaction Bob signed at PegOut time. Above the arrows, we denote the condition that unlocks the output from which the arrow starts. The arrows are blue if they represent a path taken by the operator, and red if the path is taken by one of the other committee members. The amount out_j , output of the BitVM instance $I_{i,j}$, depends on the funds still present after the BitVM dispute phase.

reclaims them in the *reimbursement* phase¹⁸. We refer to this member as the *operator*. For each operator, the remaining committee members act like *challengers*: they ensure that only honest operators can reclaim funds in the reimbursement phase by disproving any member falsely claiming to have fronted coins to Bob.

BridgeSetup. In that phase, the committee pre-signs, before the PegIn procedure is executed, specific transactions to reimburse an honest operator or punish a misbehaving one.

First, for each distinct pair of committee members denoted c_i and c_j , c_i pre-signs and forwards the transaction $\text{Slash}_{(i,j)}$ to c_j , which c_j can later publish to punish c_i upon misbehavior, as explained below. Second, for each c_i , all the committee members pre-sign the Take_i transaction that refunds c_i , as described below. If a committee member stops collaborating during setup, it is kicked out and the procedure restarts.

PegIn. Alice deposits u coins on Bitcoin via a **PegIn** transaction. This transaction has a single output that deposits the u coins into the multi-signature wallet controlled by the n committee members. The sidechain verifies the inclusion of the **PegIn** transaction in the Bitcoin blockchain using a Bitcoin light client. Once confirmed, the sidechain mints u wrapped tokens to Alice’s account.

PegOut. To withdraw the u coins, Bob first publishes a **Burn** transaction on the sidechain. Then, he constructs a **PegOut** transaction with a single output allocating $u - f_0$ coins to his Bitcoin address, where f_0 is a service fee. Bob broadcasts this **PegOut** request to the committee members. At least one committee member effectively fronts $u - f_0$ coins from their own funds to fulfill the withdrawal, in exchange for f_0 .

¹⁸To coordinate among committee members, we can define an operator schedule, for example, in a round-robin fashion.

Reimbursement. To facilitate the reimbursement of the committee member that fronted its coins to Bob, we proceed with the following construction.

Claim. Each committee member c_i can claim that they fronted coins to Bob by posting the transaction Claim_i on-chain. Claim_i takes as input an empty transaction from the committee member c_i and has an empty output that can be spent after a timelock $\text{TL}(\Delta)$ expires, along with a *connector* output, i.e., a transaction output given as input to different transactions to guarantee that only one of them will appear on-chain.

Slashing Mechanism. If c_i falsely claims to have fronted coins to Bob, any c_j can later publish on-chain $\text{Slash}_{(i,j)}$ to punish c_i . To achieve that, we employ a number of $O(n^2)$ BitVM instances as follows. For every pair of distinct committee members c_i and c_j , we consider the instance $I_{i,j}$ where c_i acts as the prover and c_j as the verifier.¹⁹

For a fixed i , the program Π , hardcoded into each instance $I_{i,j}$, verifies the following conditions: a) Bob's Burn transaction exists on the sidechain, and, b) some PegOut transaction, where operator c_i fronts the coins to Bob, exists on the Bitcoin ledger. For example, the program Π could encode a zk-SNARK verifier that verifies a proof of (1) and (2), making use of a light client of both the sidechain and the Bitcoin ledger, as shown in, e.g., [31, 10, 3].

The transaction $\text{Slash}_{(i,j)}$ takes as input: i) the connector output of Claim_i and ii) an input from any transaction where c_j wins the instance $I_{i,j}$ (i.e., any transaction where all the coins of the multi-signature are attributed to c_j) and has a single output where all the remaining coins locked in the instance $I_{i,j}$ are given to c_j .

Take. The committee member c_i that fronted coins to Bob, can finally retrieve their funds by publishing transaction Take_i on-chain. The transaction Take_i has the following inputs: i) the output of PegIn , ii) the first output of Claim_i , and iii) the connector output of Claim_i , and has a single output where the coins of the first input are transferred to c_i .

For each operator c_i , all committee members presign Take_i only if: (i) Claim_i is constructed according to the protocol, and (ii) they have received all the pre-signed transactions related to the BitVM instance $I_{i,j}$.

8.2 Security Arguments

Since the sidechain ensures PegIn succeeds, our goal is to guarantee that a successful PegIn implies a successful PegOut : if Alice mints and transfers coins to Bob, Bob can eventually reclaim the equivalent amount on Bitcoin. This relies on two properties:

Bridge Safety (no false claims): A committee member cannot falsely claim to have fronted coins to Bob. Otherwise, funds in the multisig are stolen, and future honest operators are unable to reclaim their coins.

Bridge Liveness (honest redemption): Eventually, an honest operator will front coins and reclaim them.

Safety holds because if a malicious member c_i posts Claim_i without first posting PegOut , an honest challenger c_j can initiate BitVM instance $I_{i,j}$ and post $\text{Slash}_{i,j}$, which spends the connector output of Claim_i and invalidates Take_i . Liveness holds because an honest member c_i who posts PegOut is protected: no malicious c_j can succeed with $\text{Slash}_{i,j}$, and after the Claim_i timelock, c_i can post Take_i to reclaim their funds.

¹⁹The only difference from the current protocol is that, in this case, the verifier commits the **BridgeSetup** on-chain to initiate a dispute.

8.3 Evaluation

Table 3 summarizes the transaction costs associated with each phase of the bridge protocol (to see how they are computed, refer to the full version [5, Appendix G]). Specifically, in the optimistic scenario, where all committee members behave honestly, executing an instance of our bridge protocol requires only four on-chain transactions – **PegIn**, **PegOut**, **Claim**, and **Take** – resulting in a minimal total cost of approximately \$2.04, which can be covered by the application fee f_0 paid by Bob.

On the contrary, if a challenger opens up a dispute, we have the following cases:

1. if the operator was honest, they only pay fees for **PegIn**, **PegOut**, **Claim**, and **Take** (approximately 2.04\$, covered by Bob’s fee f_0). The malicious challenger pays transaction fees associated with the execution of a BitVM instance (i.e., 534\$);
2. if the operator was illegitimately trying to reclaim coins by publishing **Claim**, they will be slashed by the challenger and will pay the cost of a BitVM instance and the fees of **Slash** and **Claim**. This amounts to 534\$.

■ **Table 3** Evaluation of a BitVM-based bridge instance w.r.t. transaction sizes and on-chain costs.

Tx	Size (<i>vB</i>)	On-chain cost (\$)
PegIn	117	0.26
PegOut	180	0.39
Claim	160	0.35
Slash	212	0.46
Take	475	1.04

9 Conclusion and Future Work

This work presents BitVM, the first protocol to enable general-purpose, trustless computation on Bitcoin without requiring consensus changes, or additional assumptions, e.g., trusted hardware or semi-trusted oracles. By combining off-chain execution with a Bitcoin-compatible dispute resolution protocol, BitVM achieves quasi-Turing completeness using existing scripting capabilities. To demonstrate the applicability of BitVM, we construct a trust-minimizing bridge and provide a prototype implementation with a concrete cost analysis.

BitVM extends the design space for Bitcoin-based applications, enabling programmable logic and verifiable off-chain computation in a trustless setting. Its ability to condition Bitcoin transactions on arbitrary program outputs opens new avenues for decentralized infrastructure anchored in Bitcoin’s security model.

Several directions remain for future work. First, while BitVM currently supports two-party interactions, generalizing the protocol to support multiparty or permissionless settings – such as decentralized oracle networks or bridges – is an important next step. Second, further reducing on-chain cost through improved dispute resolution mechanisms, such as more efficient encodings or batched verifications, could improve scalability. Third, building higher-level tooling, including compilers or domain-specific languages, would help lower the barrier to adoption and enable broader experimentation with complex BitVM-based applications.

We believe BitVM marks a foundational step in unlocking the next generation of Bitcoin-native applications.

References

- 1 tbt. URL: <https://tbt. network>.
- 2 Bitvm poc implementation. <https://github.com/BitVM/bitvm-js>, 2025.
- 3 Bob hybrid l2 technical blueprint. <https://blog.gobob.xyz/posts/bob-hybrid-l2-technical-blueprint>, 2025. Accessed: 2025-04.
- 4 What are trustless bitcoin vaults?, 2025. Accessed: 2025-11. URL: <https://babylonlabs.io/learn/what-are-trustless-bitcoin-vaults>.
- 5 Bitvm: Quasi-turing complete computation on bitcoin (full version). <https://eprint.iacr.org/2024/1995.pdf>, 2026.
- 6 Ioannis Alexopoulos, Zeta Avarikioti, Paul Gerhart, Matteo Maffei, and Dominique Schröder. BitPriv: A privacy-preserving protocol for DeFi applications on bitcoin. *Cryptology ePrint Archive*, Paper 2025/1575, 2025. URL: <https://eprint.iacr.org/2025/1575>.
- 7 Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, Giulia Scaffino, and Dionysis Zindros. Blink: An optimal proof of proof-of-work. In *Financial Cryptography and Data Security*, 2025.
- 8 Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riahi. Generalized channels from limited blockchain scripts and adaptor signatures. In *ASIACRYPT*, 2021.
- 9 Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains, 2014. URL: <https://blockstream.com/sidechains.pdf>.
- 10 Ekrem Bal, Lukas Aumayr, Atacan İyidoğan, Giulia Scaffino, Hakan Karakuş, Cengiz Eray Aslan, and Orfeas Stefanos Thyfronitis Litos. Clementine: A collateral-efficient, trust-minimized, and scalable bitcoin bridge. *Cryptology ePrint Archive*, Paper 2025/776, 2025. URL: <https://eprint.iacr.org/2025/776>.
- 11 Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. Fun with bitcoin smart contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 432–449, Cham, 2018. doi:10.1007/978-3-030-03427-6_32.
- 12 Massimo Bartoletti, Stefano Lande, and Roberto Zunino. Bitcoin covenants unchained. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 25–42, 2020. doi:10.1007/978-3-030-61467-6_3.
- 13 Massimo Bartoletti, Riccardo Marchesin, and Roberto Zunino. Secure compilation of rich smart contracts on poor UTXO blockchains. In *9th IEEE European Symposium on Security and Privacy, EuroS&P*, 2024. doi:10.1109/EUROSP60621.2024.00021.
- 14 Massimo Bartoletti and Roberto Zunino. Bitml: A calculus for bitcoin smart contracts. In *ACM CCS*, pages 83–100, 2018. doi:10.1145/3243734.3243795.
- 15 Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Draft 0.6*, 2023. URL: <https://toc.cryptobook.us/>.
- 16 Lea Salome Brugger, Laura Kovács, Anja Petkovic Komel, Sophie Rain, and Michael Rawson. Checkmate: automated game-theoretic security reasoning. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1407–1421, 2023. doi:10.1145/3576915.3623183.
- 17 Citrea. Bitcoin settlement trust-minimized btc bridge: Bitvm, 2024. Accessed: 2025-11. URL: <https://docs.citrea.xyz/technical-specs/characteristics/bitcoin-settlement-trust-minimized-btc-bridge/bitvm>.
- 18 Poulami Das, Lisa Ekey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. FastKitten: Practical smart contracts on bitcoin. In *USENIX Security 19*, 2019.
- 19 Contributor DLC. Discreet log contracts: An overview, 2018. URL: <https://adiabat.github.io/dlc.pdf>.
- 20 Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In *ACM CCS*, pages 949–966, 2018. doi:10.1145/3243734.3243856.
- 21 Esolangs. Addleq, 2025. URL: <https://esolangs.org/wiki/Addleq>.

- 22 Ethereum Foundation. State channels, 2024. Accessed: 2025-04. URL: <https://ethereum.org/en/developers/docs/scaling/state-channels/>.
- 23 FairGate Labs. Bitvmx & bitvm, 2024. Accessed: 2025-11. URL: <https://fairgate.io>.
- 24 Tommaso Frassetto, Patrick Jauernig, David Koisser, David Kretzler, Benjamin Schlosser, Sebastian Faust, and Ahmad-Reza Sadeghi. Pose: Practical off-chain smart contract execution. In *NDSS Symposium*. Internet Society, 2023. doi:10.14722/ndss.2023.23118.
- 25 Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. *J. ACM*, 71(4), August 2024. doi:10.1145/3653445.
- 26 Sanjam Garg, Dimitris Kolonelos, Mikhail Sergeevitch, Srivatsan Sridhar, and David Tse. BABE: Verifying proofs on bitcoin made 1000x cheaper. Cryptology ePrint Archive, Paper 2026/065, 2026. URL: <https://eprint.iacr.org/2026/065>.
- 27 Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1353–1370, 2018. URL: <https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner>.
- 28 Leslie Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, SRI International, October 1979. URL: <https://www.microsoft.com/en-us/research/publication/constructing-digital-signatures-one-way-function/>.
- 29 Robin Linus. Bitvm: Compute anything on bitcoin, 2023. Accessed: 2025-11. URL: <https://bitvm.org/bitvm.pdf>.
- 30 Robin Linus, Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, Andrea Pelosi, Orfeas Thyfronitis Litos, Christos Stefo, David Tse, and Alexei Zamyatin. Bridging bitcoin to second layers via BitVM2. In *35th USENIX Security Symposium (USENIX Security 26)*, 2026.
- 31 Robin Linus, Lukas Aumayr, Alexei Zamyatin, Andrea Pelosi, Zeta Avarikioti, and Matteo Maffei. BitVM2: Bridging bitcoin to second layers, 2024. Accessed: 2025-04. URL: https://bitvm.org/bitvm_bridge.pdf.
- 32 Eric Lombrozo and Pieter Wuille. Bip 141, segregated witness, 2015. Accessed: 2025-04. URL: <https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki>.
- 33 Varun Madathil, Sri AravindaKrishnan Thyagarajan, Dimitrios Vasilopoulos, Lloyd Fournier, Giulio Malavolta, and Pedro Moreno-Sanchez. Cryptographic oracle-based conditional payments. In *NDSS Symposium*, 2023. doi:10.14722/ndss.2023.24024.
- 34 Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989. doi:10.1007/0-387-34805-0_21.
- 35 Flavius Plesu. Turing completeness in evm machines, 2023. Accessed: 2025-04. URL: <https://www.flavius.io/media/turing-completeness-in-evm-machines>.
- 36 Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.
- 37 Sophie Rain, Georgia Avarikioti, Laura Kovács, and Matteo Maffei. Towards a game-theoretic security analysis of off-chain protocols. In *IEEE CSF*, pages 107–122, 2023. doi:10.1109/CSF57540.2023.00003.
- 38 Sovryn. Bitcoinos, 2024. Accessed: 2025-11. URL: <https://sovryn.com/bitcoins>.
- 39 Apostolos Tzinas, Srivatsan Sridhar, and Dionysis Zindros. On-chain timestamps are accurate. Cryptology ePrint Archive, Paper 2023/1648, 2023. URL: <https://eprint.iacr.org/2023/1648>.
- 40 Robin Linus Woll, Ioannis Alexopoulos, Lukas Aumayr, Zeta Avarikioti, Matteo Maffei, and David Tse. BitVM3: Efficient bitcoin bridges via garbled circuits. In *ACM Conference on Computer and Communications Security (CCS)*, 2026.
- 41 Pieter Wuille, Jonas Nick, and Anthony Towns. Bip 0341, taproot: Segwit version 1 spending rules, January 2020. URL: https://en.bitcoin.it/wiki/BIP_0341.