BitML: A Calculus for Bitcoin Smart Contracts

Massimo Bartoletti
University of Cagliari
bart@unica.it

Roberto Zunino
University of Trento
roberto.zunino@unitn.it

ABSTRACT

We introduce BitML, a domain-specific language for specifying contracts that regulate transfers of bitcoins among participants, without relying on trusted intermediaries. We define a symbolic and a computational model for reasoning about BitML security. In the symbolic model, participants act according to the semantics of BitML, while in the computational model they exchange bitstrings, and read/append transactions on the Bitcoin blockchain. A compiler is provided to translate contracts into standard Bitcoin transactions. Participants can execute a contract by appending these transactions on the Bitcoin blockchain, according to their strategies. We prove the correctness of our compiler, showing that computational attacks on compiled contracts are also observable in the symbolic model.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; **Formal security models**; **Security protocols**;

KEYWORDS

Bitcoin; smart contracts; process calculi

ACM Reference Format:
Massimo Bartoletti and Roberto Zunino. 2018. BitML: A Calculus for Bitcoin Smart Contracts. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3243734.3243795

1 INTRODUCTION

Cryptocurrencies like Bitcoin and Ethereum have revived the idea of *smart contracts* — agreements between untrusted parties that can be automatically enforced without a trusted intermediary [58]. These agreements regulate cryptocurrency exchanges among participants: for instance, a lottery collects bets from players, determines the winner in a fair manner, and then transfers the pot to the winner.

Disintermediation is made possible by the *blockchain*, a public, append-only record of transactions, and by the *consensus protocol* followed by the nodes to update the blockchain [27]. The execution of smart contracts relies on the blockchain to log all the participants' moves; further, the underlying logic of transactions is exploited to enable all and only the moves permitted by the contract. The consensus protocol is used to consistently update the blockchain: suitable economic incentives ensure that the nodes of the network have the same view of the blockchain. In this way, the state of each contract (and consequently, the asset of each user) is uniquely determined by the sequence of its transactions on the blockchain.

Smart contracts have different incarnations, depending on the platform on which they are based. In Ethereum, they are expressed as programs in a Turing-equivalent bytecode language. Any user can publish a contract on the blockchain. This makes the contract available to other users, who can then run it by calling its functions (concretely, by publishing suitable transactions on the blockchain). Such openness comes at the price of a wide attack surface: attackers may exploit vulnerabilities in the implementation of contracts, or may publish themselves Trojan-horses with hidden vulnerabilities, to steal or tamper with the assets controlled by contracts. Indeed, a series of vulnerabilities in Ethereum contracts [13] have caused losses in the order of hundreds of millions of USD [3, 5, 6].

Unlike Ethereum, Bitcoin does not provide a language for smart contracts: rather, in literature they are expressed as cryptographic protocols where participants send/receive/sign messages, verify signatures, and put/search transactions on the blockchain [14]. Lotteries [10, 19, 22, 47], gambling games [42], micro-payment channels [31, 48, 54], contingent payments [17, 32, 46], and more general fair multi-party computations [11, 41] witness the variety of smart contracts supported by Bitcoin.

Describing smart contracts at this level of abstraction is complex and error-prone. Indeed, establishing the correctness of a smart contract requires to prove the *computational security* of a cryptographic protocol, where — besides the usual primitives — participants can craft Bitcoin transactions and interact with the Bitcoin network. Further, these protocols often rely on advanced features of Bitcoin (e.g., transaction scripts, signature modifiers, segregated witnesses), whose actual behaviour relies on low-level implementation details. The task of proving the security of such kind of protocols requires the skills of expert cryptographers, and even in this case it is a significant effort. By contrast, working in an high-level *symbolic* model would relieve smart contract programmers from (most of) this burden, since the much higher level of abstraction would allow security proofs to be carried out with automatic tools.

Contributions. We introduce BitML (after "Bitcoin Modelling Language"), a domain-specific language for Bitcoin smart contracts. BitML is a process calculus, with primitives to stipulate contracts and to exchange currency according to the contract terms. In this respect, BitML departs from the current practice of representing Bitcoin contracts as cryptographic protocols: rather, BitML pioneers the "contracts-as-programs" paradigm for Bitcoin, by completely abstracting from Bitcoin transactions and cryptographic details. Despite the high level of abstraction, BitML can express most of the Bitcoin smart contracts proposed so far [14], e.g. escrow services, timed commitments, lotteries, gambling games, etc. The operational semantics of BitML allows for reasoning about the behaviour of

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243795

these contracts in a *symbolic* setting, where the underlying cryptography and Bitcoin machinery are abstracted away.

One of our main contributions is a compiler to translate BitML contracts into standard Bitcoin transactions. Participants can perform the contract actions by publishing the corresponding transactions on the blockchain. The crucial technical challenge is to guarantee the correctness of the compiler, i.e. that the "symbolic" execution of the contract matches the "computational" one performed on Bitcoin. This correspondence must hold also in the presence of computational adversaries: otherwise, attacks at the Bitcoin level could be unobservable at the level of the symbolic semantics.

We establish the correctness of the BitML compiler through a *computational soundness* theorem [8]. More specifically, we prove that if honest participants use compiler-generated transactions, then the actual Bitcoin executions resulting from their interaction with computational adversaries will have a symbolic counterpart as a BitML execution (with overwhelming probability). Basically, this implies that computational attacks to compiled contracts in Bitcoin are also observable in the symbolic semantics of BitML. A practical consequence of this result is that proofs of trace-based security properties carried out in the symbolic model can be lifted *for free* to the computational model. This result is crucial, since it enables the development of analysis and verification techniques *at the symbolic level*, which would be much more burdensome to obtain at the (far more concrete) computational level.

A technical report containing the full formal machinery of our work is available online [20].

2 OVERVIEW OF THE APPROACH

In this section we overview our approach, introducing its main components, and discussing the relations among them. In particular, we provide a gentle introduction to BitML, illustrating its expressiveness through a series of examples. We then discuss the main results of the paper, illustrating their practical consequences.

BitML in a nutshell. In BitML, contracts allow participants to interact according to the following workflow. First, a participant broadcasts a *contract advertisement* $\{G\}C$. The component C is the contract, which specifies the rules to transfer bitcoins (฿) among participants. The component G is a set of *preconditions* to its execution: roughly, it requires participants to deposit some ฿, either upfront or during the contract execution, and to commit to some secrets. Participants can then choose whether to accept the advertisement, or not. When all the involved participants have accepted $\{G\}C$, satisfying its preconditions, the contract C becomes stipulated. Only at this point, participants can transfer the deposited funds by acting as prescribed by C. Once C is stipulated, it starts its execution with a *balance*, initially set to the sum of the deposits in its advertisement. The execution of C will affect this balance, when participants deposit/withdraw funds to/from the contract.

As a first example, assume a buyer A who wants to buy an item from a seller B. The participants want to use a contract to ensure that B will get paid if and only if A gets her item. Assuming the cost of the item is 1฿, the contract precondition:

$$G \;=\; \mathsf{A} : !\; 1฿ \,@\, x$$

requires A to provide a 1฿ deposit (where x is the deposit name), which will be transferred to B only after A's consent. The contract has two mutually exclusive clauses (separated by +):

$$\mathit{PayOrRefund} \;=\; \mathsf{A} : \mathtt{withdraw}\ \mathsf{B} \;+\; \mathsf{B} : \mathtt{withdraw}\ \mathsf{A}$$

The first clause allows B to withdraw 1฿ from the contract, if A provides her authorization (denoted by $\mathsf{A} : \cdots$). Instead, the second clause allows A to get back her deposit upon B's authorization (e.g., in case B acknowledges a problem with the shipment).

Note that the above contract gives little guarantees when the participants dishonestly deny their authorization: in particular, A can receive the item and then prevent B from withdrawing the payment, while B can freeze A's deposit without shipping the item. A less naïve contract should guarantee that, even if A or B are dishonest, exactly one of them will be able to redeem the deposit. To ensure this property, we resort to a mediator M who resolves disputes between A and B. Assuming that the mediator takes a fee of 0.1฿ (cut down from A's deposit), we can craft the following:

$$\mathit{Escrow} = \mathit{PayOrRefund} + \mathsf{A} : \mathit{Resolve}_{0.1,0.9} + \mathsf{B} : \mathit{Resolve}_{0.1,0.9}$$

$$\mathit{Resolve}_{v,v'} = \mathtt{split}(v฿ \to \mathtt{withdraw}\ \mathsf{M} \;\mid\; v'฿ \to \mathsf{M} : \mathtt{withdraw}\ \mathsf{A} + \mathsf{M} : \mathtt{withdraw}\ \mathsf{B})$$

Besides the two clauses in *PayOrRefund*, the contract *Escrow* features two additional clauses, which allow A and B to trigger the dispute resolution, specified by *Resolve*. The two parallel clauses therein split the 1฿ deposit in two parts: 0.1฿ go to the mediator, while 0.9฿ are assigned either to A and B, depending on M's choice.

BitML contracts feature other primitives besides those seen so far: for instance, they can express time constraints, and allow participants to choose/reveal secrets. We show these features to model a *timed commitment* protocol [11, 26, 35, 57]. There, a participant A wants to choose a secret, and reveal it after some time t — guaranteeing that the revealed value is the chosen secret. We force A to pay to another participant B a penalty of 1฿ if A does not reveal the secret within t. In the contract precondition, A declares a deposit of 1฿, and a secret with name a. The contract is the following:

$$\mathit{TC} \;=\; (\mathtt{reveal}\ a.\ \mathtt{withdraw}\ \mathsf{A}) \;+\; (\mathtt{after}\ t : \mathtt{withdraw}\ \mathsf{B})$$

Only A can choose the first clause, by revealing a. When doing so, A can take her 1฿ deposit back. After the deadline t, B can choose the second clause, and collect A's penalty. Before t, A has the option to reveal a (avoiding the penalty), or to keep it secret (paying the penalty). As a borderline case, if A reveals a after t, a race condition occurs: the first one who makes a step gets the money.

Timed commitment contracts like the one above are the basis upon which constructing complex contracts which distribute bitcoins according to values chosen by participants (e.g., gambling games, lotteries, etc.). For instance, consider a simple "odds and evens" game between two players. The contract preconditions require A and B to commit to one secret each (a and b, respectively), and to put a deposit of 3฿ each (1฿ as a bet, and 2฿ as a penalty in

case of dishonest behaviour). The contract is the following:

$$\begin{aligned} \mathit{OddsEvens} = \texttt{split}(&\\ &2\text{฿} \to \texttt{reveal}\ b\ \texttt{if}\ 0 \le |b| \le 1.\ \texttt{withdraw}\ \mathsf{B}\\ &\quad + \texttt{after}\ t : \texttt{withdraw}\ \mathsf{A}\\ &\mid 2\text{฿} \to \texttt{reveal}\ a.\ \texttt{withdraw}\ \mathsf{A}\ +\ \texttt{after}\ t : \texttt{withdraw}\ \mathsf{B}\\ &\mid 2\text{฿} \to \texttt{reveal}\ a\ b\ \texttt{if}\ |a| = |b|.\ \texttt{withdraw}\ \mathsf{A}\\ &\quad + \texttt{reveal}\ a\ b\ \texttt{if}\ |a| \neq |b|.\ \texttt{withdraw}\ \mathsf{B}\ ) \end{aligned}$$

The balance is split in three parts. Player B must reveal b by the deadline t; otherwise, A can redeem B's penalty (as in the timed commitment). Similarly, A must reveal a. To determine the winner we compare the *lengths* of the secrets, in the third clause of `split`. The winner is A if the secrets have the same length, otherwise it is B. Checking that b's length is either 0 or 1 (in the first clause) is needed to achieve fairness: indeed, B can increase his probability to win 2฿ in the third clause by choosing a secret with length $N > 1$. However, doing so would make B lose his 2฿ deposit, so overall B's *average* payoff would be negative. So, a rational B would then choose a secret of length 0 or 1 — as well as a rational A, who otherwise decreases her probability to win. When both lengths are 0 or 1, A and B can redeem their 2฿ penalty, and have a 1/2 probability to win, if at least one of them chooses the length uniformly.

The last primitive we present is $\texttt{put}\ x$, which allows a participant to provide a "volatile" deposit x after stipulation. We illustrate it in a variant of *Escrow*, where both A and B contribute to the fee for the mediator, only if they do not agree within *PayOrRefund*:

$$\begin{aligned} \mathit{EscrowPut} = \mathit{PayOrRefund} + \texttt{after}\ t : \texttt{withdraw}\ \mathsf{B}\\ + \texttt{put}\ x\ .\ (\texttt{put}\ y\ .\ \mathit{Resolve}_{0.2,1} + \texttt{after}\ t' : \texttt{withdraw}\ \mathsf{A}) \end{aligned}$$

As a precondition, we assume that A and B declare additional volatile 0.1฿ deposits (respectively, x and y). If they disagree, A has to pay her deposit before time t, otherwise B can withdraw A's 1฿ deposit. The same for B, which must pay his 0.1฿ deposit before $t' > t$. After that, they invoke *Resolve*, paying 0.2฿ to the mediator, and assigning 1฿ to the winner of the dispute.

Symbolic model. To reason about the security of BitML, we introduce two security models: a symbolic model and a computational one. The symbolic model is based on the semantics of BitML, which abstracts from Bitcoin and its blockchain. In this model, participants are represented as strategies (formally, PPTIME algorithms) which allow them to choose which actions to perform, in any given state. Each honest participant has its own strategy, while the dishonest ones are collectively represented as a single adversarial strategy. Strategies can read the current BitML *configuration* — a public, shared state — and output actions which determine the next state. Further, the adversary strategy can schedule the participants' moves, eavesdrop messages, and impersonate other participants. Within a configuration, we can find e.g. *deposits* of the form $\langle \mathsf{A}, v\text{฿}\rangle$, modelling the ownership of v฿ by participant A. Participant can freely split or gather their own deposits, or transfer them to other participants. Further, deposits can be spent to stipulate contracts: technically, stipulation creates an *active contract*, i.e. a term of the form $\langle C, v\text{฿}\rangle$, with an initial balance v฿ amounting to the sum of all the spent deposits. The contract C determines how the balance of v฿ can be distributed among participants, depending on their actual interaction. Contracts have an operational semantics, describing all their possible transitions: participants strategies (either honest or adversarial) can only choose among the transitions allowed by the semantics. For instance, a possible symbolic run of *Escrow* is the following (omitting the stipulation and the intermediate steps):

$$\langle \mathit{Escrow}, 1\text{฿}\rangle \ \to^*\ \langle \mathsf{A}, 0.9\text{฿}\rangle \mid \langle \mathsf{M}, 0.1\text{฿}\rangle$$

which models the case where the mediator has resolved a dispute in favour of A (there are also runs where the dispute is won by B, or the mediator is not invoked). The timed commitment *TC* has two kinds of runs, according to A's strategy: those where A reveals the secret, and those where B redeems the deposit. For instance:

$$\langle \mathit{TC}, 1\text{฿}\rangle \mid t_0 \ \to^*\ \langle \mathsf{A}, 1\text{฿}\rangle \mid \mathsf{A} : a\#N \mid t_0$$

is an run where A has revealed the secret (represented by the term $\mathsf{A} : a\#N$, where N is a's length) at time $t_0 < t$. Instead, in the run:

$$\langle \mathit{TC}, 1\text{฿}\rangle \mid t_0 \ \to^*\ \langle \mathsf{B}, 1\text{฿}\rangle \mid t_0 + \delta$$

with $t_0 + \delta > t$, A has not revealed the secret, and B has collected A's 1฿ penalty. Finally, *OddsEvens* features symbolic runs where either A or B redeem the winnings, and they reveal or not their secrets. For instance, in the following run (where $t_0 < t$):

$$\langle \mathit{OddsEvens}, 6\text{฿}\rangle \mid t_0 \ \to^*\ \langle \mathsf{A}, 2\text{฿}\rangle \mid \langle \mathsf{B}, 4\text{฿}\rangle \mid \mathsf{A} : a\#0 \mid \mathsf{B} : b\#1 \mid t_0$$

players have revealed their secrets (and redeemed the 2฿ deposits), and B has won the pot, since the lengths of the secrets are different.

Compiling BitML to Bitcoin. We design a compiler that, given a BitML contract, outputs a set of Bitcoin transactions through which the contract can be actually executed on Bitcoin. Our compiler only relies on standard features of Bitcoin transactions: for instance, the `withdraw` primitive relies on signature verification, `split` relies on transactions with multiple outputs, and `reveal` exploits the hash opcode of Bitcoin scripts. Sequencing and choice are obtained, respectively, by transaction cascading and by the absence of double-spending. We show several examples of compilation in Section 8.

A crucial question is: *how to preserve security properties of a BitML contract once it is compiled into Bitcoin?* Indeed, lifting symbolic security to computational security is a necessary result in order to justify the adoption of high-level languages for smart contracts: in the absence of such result, proving that a contract is secure in the symbolic model would not guarantee its security under a computational attacker. Proving such preservation result is challenging, as the symbolic model is a substantially higher-level than what can actually happen in Bitcoin. For instance, while the executions of contracts strictly follow the BitML semantics (so, the possible configurations are pre-determined), in Bitcoin adversaries can append arbitrary transactions to the blockchain (not necessarily those obtained by the compiler), crafting them with data in their knowledge, possibly sniffed over the network.

Computational model. In order to reason about the behaviour of participants in Bitcoin, we introduce a computational model, where attackers are only subject to the usual restrictions of standard computational models (i.e. they can only manipulate bitstrings using PPTIME algorithms). As in the symbolic model, also computational participants are rendered as strategies: these strategies can listen to network traffic and scan the blockchain, in order to decide their next actions. As an action, a participant can append a

transaction to the blockchain: for this to happen, the bitstring generated by a strategy must be the encoding of a transaction which redeems some unspent transaction outputs on the blockchain. Another action is broadcasting a bitstring: this is used for off-chain communications, e.g., when participants need to exchange signatures, or to reveal their secrets. The adversary strategy defines the behaviour of the dishonest participants, and acts as a scheduler for all the participants' moves, as in the symbolic model. The main difference, here, is that computational actions are bitstrings, instead of symbolic terms. Computational adversaries can obtain these bitstrings from their knowledge (e.g., sniffed messages), with the only limitation of crafting them through PPTIME algorithms. For simplicity, our model slightly abstracts from the actual low-level behaviour of Bitcoin (e.g., we assume the blockchain to be immutable and without forks, and we neglect transaction fees).

Relating symbolic and computational security. The main result of this paper is a *computational soundness* theorem [8]: it states that, with overwhelming probability, runs in the computational model have a corresponding run in the symbolic model. Therefore, computational attacks can be observed in the symbolic model. More precisely, the theorem states that we can securely execute BitML contracts on Bitcoin according to the following workflow. First, each honest participant chooses her symbolic strategy, which drives the stipulation and execution of contracts according to the symbolic semantics. Then, they translate their strategies to computational ones; in this step, we leverage our compiler to map the involved contracts into Bitcoin transactions. Finally, participants execute their computational strategies, being subject to computational adversaries. Our computational soundness result ensures that computational strategies admit no attacks, unless the original symbolic strategies also admit the same attack. To state computational soundness, we define a correspondence relation (called *coherence*) between runs of contracts in the two models. Intuitively, (symbolic) deposits $\langle A, v\text{฿}\rangle$ correspond to (computational) unspent transaction outputs for v฿ which can be redeemed using only A's signature. Active contracts $\langle C, v\rangle$ also correspond to unspent transaction outputs, but they involve more complex redeem conditions (e.g., they may require to provide a signature from *all* the contract participants, or to reveal a secret having a certain length). Performing a transition in the symbolic semantics of a contract C corresponds to appending to the blockchain one of the transactions obtained by compiling C.

Practical consequences of computational soundness. To illustrate the practical applications of our results, recall the *OddsEvens* contract introduced before. Assume that A's symbolic strategy is to (i) stipulate the contract, depositing 3฿ and committing to a randomly chosen secret a of length 0 or 1; (ii) reveal her secret at a certain time $t' < t$; (iii) withdraw her deposit at time t'; (iv) collect the pot, if she is the winner. According to the symbolic semantics, in all the runs of *OddsEvens* which are *conformant* to A' strategy (and to any adversarial strategy), after time t', A will receive at least 2฿ (i.e. her deposit) from the contract. Computational soundness guarantees that the same holds in the computational model, i.e. when executing the compiled contract on Bitcoin. Note that proving this result directly in the computational model is inconvenient, as the adversary — not being constrained to follow the structure of the contract — can play any sequence of actions that respects the computational model, and the consistency of the Bitcoin blockchain (where he can also append transactions crafted by himself, not obtained by the BitML compiler). In this setting, proving security against all adversaries would require to cope with a universal quantification over all possible PPTIME algorithms. By contrast, proving security in the symbolic model is significantly simpler: a verification algorithm should check if the desired property holds in all the reachable configurations in the contract runs conformant with A's strategy. This simplification is similar to the one obtained when reasoning about the security of cryptographic protocols in symbolic models, instead of computational ones [8].

3 RELATED WORK

The first proposal to implement smart contracts on Bitcoin dates back (at least) to 2012 [1], and the first scientific paper to 2013 [11]. Since then, the research on smart contracts has evolved along different directions: (i) studying contracts that can be run directly on Bitcoin; (ii) increasing the expressiveness of contracts through Bitcoin extensions; (iii) developing high-level languages for Bitcoin contracts; (iv) studying new blockchain infrastructures for smart contracts. Below we briefly survey the literature along these lines.

Bitcoin smart contracts. Basic smart contracts which transfer bitcoins according to the external state [1, 2] can be implemented using multi-signature transactions. Timed commitments for Bitcoin were originally introduced by [11], and then used to implement multiparty computations, like lotteries. The lottery in [11] requires each player to deposit a collateral which grows quadratically with the number of players. Subsequent works proposed Bitcoin extensions which allow for lotteries without collaterals. More general forms of fair multiparty computations on "pure" Bitcoin, exploiting SegWit [43], were proposed in [10, 22, 41]. Contingent payments for Bitcoin (i.e., contracts which allow to sell solutions for a class of NP problems) were introduced in [17, 46].

Extensions of the Bitcoin scripting language. Other works proposed extensions of Bitcoin to enhance the expressiveness of smart contracts. In [49] the Bitcoin scripting language is extended with *covenants*, a construct that can constrain the structure of the redeeming transaction. Another implementation of covenants is proposed in [53], exploiting a (currently disabled) opcode to concatenate arbitrary data, and introducing a new opcode to verify signatures against it. Covenants enable contracts that implement *vaults*, i.e. protocols which scatter a money transfer along a sequence of transactions, giving the ability to the owner of the vault to abort the transfer if he detects misbehaviour. More generally, recursive covenants allow to implement a state machine through a sequence of transaction that store its state. The collaterals of multiparty lotteries can be eliminated through Bitcoin extensions: e.g., [19] requires input malleability (i.e. the possibility of not signing any input), while [47] requires a new opcode that checks if the redeeming transaction belongs to a predetermined set. The work [32] proposed a contingent payment protocol that does not rely on zero-knowledge proofs, but instead requires a new opcode to check if the two top elements of the stack are a valid key pair. The work [42] captures general multiparty, interactive, stateful computations by exploiting a new opcode (similar to the one in [53]), to check signatures for arbitrary messages.

Languages for Bitcoin smart contracts. Only a few languages for Bitcoin contracts have been proposed so far. TypeCoin [30] allows to model the updates of a state machine as affine logic propositions. Users "run" this machine by putting transactions on the blockchain, with the guarantee that only legit updates can be performed. A downside of [30] is that liveness is guaranteed only when participants cooperate, i.e., an adversary can prevent the others from completing the contract. Note instead that in BitML, honest participants can always make a contract progress. The other languages we are aware of, IVY [4], BALZaC [7] and Simplicity [52], are high-level alternatives to the Bitcoin scripting language, that can be compiled into Bitcoin scripts. In order to implement a smart contract using these languages, one still needs to design it as a protocol involving message exchanges and transactions (although with more readable scripts). Note that these languages do not allow to describe the whole contract, but only the *individual* transactions used in the associated protocol. Compared to these approaches, BitML has two main advantages: first, it can express the *whole* contract within a single term; second, it relieves the designer from the burden of Bitcoin transactions, which, instead, can be automatically generated by our compiler. The loss of expressiveness caused by the abstraction from low-level Bitcoin details (discussed in Section 11), is repaid by a gain of elegance in the specifications of contracts, and by the simplification of upcoming verification techniques.

Smart contracts beyond Bitcoin. After Bitcoin, other platforms and languages for smart contracts have been created [18]. Currently, the most popular ones are Ethereum [28] (in the permissionless setting) and Hyperledger Fabric [9] (in the permissioned one). The recent attacks on Ethereum contracts have given rise to extensive research on how to make it more secure. A few papers study EVM, the bytecode language which is the target of compilation of Ethereum contracts. Among them, [44] formalises the semantics of EVM, and develops a symbolic execution of EVM contracts, to detect some vulnerability patterns. Another approach based on the analysis of vulnerability patterns on dependency graphs is pursued by [59], which develop a tool called Securify. A more detailed formalisation of EVM (validated against the official Ethereum test suite) is in [37], which also proposes a set of general security properties relevant for avoiding classic vulnerabilities in Ethereum contracts. The work [36] develops EtherTrust, a framework for the static verification of Ethereum smart contracts at the EVM level, which can establish the absence of re-entrancy vulnerabilities. Also [23] detects vulnerabilities of Ethereum contracts, by translating Solidity and EVM code into F* [56]. Further, given a Solidity program and an alleged compilation of it into EVM bytecode, [23] verifies that the two pieces of code have equivalent behaviours. The work [38] uses the Isabelle/HOL proof assistant [51] to verify the EVM obtained by compiling the Solidity code of "Deed", a contract which is part of the Ethereum Name Service. In particular, it proves that, upon an invocation of the contract, only its owner can decrease the balance. The work [55] proposes Scilla, a strongly typed intermediate language where contracts are represented as Communicating Automata. Compared to EVM, Scilla is more structured: this simplifies formal reasoning, and makes contracts more amenable to verification. Ongoing work aims at a Coq formalization of Scilla.

Other papers propose domain-specific languages for Ethereum contracts. Among them, [45] represents smart contracts as finite state automata, where state transitions can be constrained according to contract variables and inputs; a tool is provided to translate these automata into Solidity code. The work [24] compiles to Solidity a fragment of the language for financial contracts introduced by [39]. While the previous works address qualitative properties of contracts, the work [29] develops a framework for their *quantitative* analysis, by transforming contracts (specified in an abstract language) into state-based games. This allows one to compute the worst-case guaranteed utility resulting from interacting with a contract, which can be helpful to detect vulnerabilities.

4 THE BITML CALCULUS

We assume a set Part of *participants*, ranged over by A, B, . . ., and we denote with Hon ⊆ Part a non-empty set of *honest* participants. We also assume a set of names, of two kinds: $x, y, \ldots$ denote *deposits* of ₿, while $a, b, \ldots$ denote *secrets*. We denote with $\vec{x}$ a finite sequence of deposit names, and we adopt a similar notation for sequences of other kinds.

Definition 1 (Contract preconditions).

G ::=	$A : ? \, v \, @ \, x$	volatile deposit of v₿, expected from A
	$\mid A : ! \, v \, @ \, x$	persistent deposit of v₿, expected from A
	$\mid A : \texttt{secret} \, a$	committed secret by A
	$\mid G \mid G$	composition ◇

The precondition $A : ! \, v \, @ \, x$ requires A to own v₿ in a deposit x, and to spend it for stipulating a contract C. Instead, $A : ? \, v \, @ \, x$ only requires A to pre-authorize the spending of x. Since x is not spent upfront, there is no guarantee that v₿ will be available when C demands x, as A can spend it for other purposes. Finally, $A : \texttt{secret} \, a$ requires A to generate a random nonce a, and commit to it before C starts. During the execution of C, A can choose whether to reveal a to the other participants, or not.

Definition 2 (Contracts). The syntax of contracts is in Figure 1, We abbreviate $\texttt{put} \, \vec{x} \, \& \, \texttt{reveal} \, \vec{a} \, \texttt{if} \, p$ as: (i) $\texttt{put} \, \vec{x}$ when $\vec{a}$ is empty and p is *true*, (ii) $\texttt{reveal} \, \vec{a} \, \texttt{if} \, p$ when $\vec{x}$ is empty, (iii) τ when $\vec{x}$ and $\vec{a}$ are empty and p is *true*, and (iv) we omit "$\texttt{if} \, p$" when the predicate p is true. We denote with 0 the empty sum. In guarded contracts, we assume that the order of decorations is immaterial, e.g., we consider $\texttt{after} \, t : A : B : D$ equivalent to $B : A : \texttt{after} \, t : D$. ◇

A contract C is a *choice* among branches. Intuitively, a branch D performs an action, and possibly proceeds with a continuation C'. The action $\texttt{put} \, \vec{x} \, \& \, \texttt{reveal} \, \vec{a} \, \texttt{if} \, p$ atomically performs the following: (i) spend all the volatile deposits $\vec{x}$, adding their values to the current balance; (ii) check that all the secrets $\vec{a}$ have been revealed, and that they satisfy the predicate p. The guarded contract $\texttt{split} \; v_1 \to C_1 \mid \cdots \mid v_n \to C_n$ divides the contract into n contracts C_i, each one with balance v_i. The sum of the v_i must be equal to the current balance. The prefix $\texttt{withdraw} \; A$ transfers the whole balance to A (to transfer only a part of it, one can perform a split). Note that, when enabled, the above actions can be fired by anyone at anytime. To restrict *who* can execute a branch and *when*, one can use the decoration $A : D$, which requires the authorization of A, and the decoration $\texttt{after} \, t : D$, which requires to wait until time t.

$C ::= \sum_{i \in I} D_i$	contract
$D ::=$	guarded contract
$\texttt{put}\ \vec{x}\ \&\ \texttt{reveal}\ \vec{a}\ \texttt{if}\ p.\ C$	collect deposits $\vec{x}$ and secrets $\vec{a}$
$\mid \texttt{withdraw}\ A$	transfer the balance to A
$\mid \texttt{split}\ \vec{v} \to \vec{C}$	split the balance ($\lvert\vec{v}\rvert = \lvert\vec{C}\rvert$)
$\mid A : D$	wait for A's authorization
$\mid \texttt{after}\ t : D$	wait until time t

$p ::=$	predicate
$true$	truth
$\mid p \wedge p$	conjunction
$\mid \neg p$	negation
$\mid E = E$	equality
$\mid E < E$	less than

$E ::=$	arithmetic expression
N	32-bit constant
$\mid \lvert a \rvert$	length of a secret
$\mid E + E$	addition
$\mid E - E$	subtraction

Figure 1: Syntax of BitML contracts.

Definition 3 (Contract advertisement). A contract advertisement is a term $\{G\}C$ satisfying the following conditions: (i) the names in G are distinct; (ii) each name in C occurs in G; (iii) the names in $\texttt{put}\ \vec{x}\ \&\ \texttt{reveal}\ \vec{a}\ \texttt{if}\ p$ are distinct; and, each name in p occurs in $\vec{a}$; (iv) each A in $\{G\}C$ has a persistent deposit in G. ⋄

The last condition will be used to guarantee that the contract is stipulated only if *all* the involved participants give their authorizations. Indeed, in order to transform a contract advertisement $\{G\}C$ into an active contract, our semantics requires only the authorizations of participants with persistent deposits in G. So, condition (iv), makes the participants involved in $\{G\}C$ equal to those with persistent deposits in G, causing the stipulation to happen only when everyone agrees. Requiring exactly the authorizations to spend persistent deposits in the symbolic semantics is key to implement contract stipulation in Bitcoin: indeed, to record that a contract has been stipulated, in Bitcoin one has to append a suitable transaction, say T_{init}. When doing this, the Bitcoin network cannot check conditions corresponding to committing to a secret, or to pre-authorize a volatile deposit, since these actions are performed *off-chain*. The only condition that can be checked is that persistent deposits are spent contextually with appending T_{init}: this can be obtained by setting the inputs of T_{init} to these deposits. Note also that condition (iv) is not restrictive in practice: we can craft a contract that allows a participant A to deposit just a small fraction of bitcoin, and then immediately transfer it back to A through a `split`.

Semantics We introduce a reduction semantics of BitML. Because of space limitations, here we provide the underlying intuitions, relegating the full formalisation to Appendix A and [20]. The rules of the semantics are grouped into four sets: (i) rules for managing deposits; (ii) rules for advertising contracts and stipulating them; (iii) rules for executing active contracts; (iv) rules for handling time. The untimed rules follow a common pattern, in order to perform an operation: first, the involved participants give their authorization; then, the operation is actually performed. Both cases are rendered as transitions in the semantics. Transitions are decorated with labels, which describe the performed actions. For simplicity, we ignore these labels in our informal description below.

The configurations of the semantics contain the following terms: (i) *contract advertisements* $\{G\}C$ represent a contract which has been proposed, but not stipulated yet; (ii) *active contracts* $\langle C, v\rangle_x$ represent a stipulated contract, holding a current balance of v฿. The name x uniquely identifies the active contract; (iii) *personal deposits* $\langle A, v\rangle_x$ represent a fund of v฿ owned by A, and with unique name x; (iv) *authorizations* $A[\chi]$ represent the consent of A to perform some operation χ; (v) *committed secrets* $\{A : a\#N\}$, represent A who has committed a random nonce a of (secret) length N, by broadcasting its hash $H(a)$; (vi) *revealed secrets* $A : a\#N$ represent the fact that A has revealed her secret a (hence, its length N).

Definition 4 (Configurations). The syntax of configurations is:

$\Gamma ::= 0$	empty
$\mid \{G\}C$	contract advertisement
$\mid \langle C, v\rangle_x$	an active contract containing v฿
$\mid \langle A, v\rangle_x$	a deposit of v฿ redeemable by A
$\mid A[\chi]$	authorization of A to perform χ
$\mid \{A : a\#N\}$	committed secret of A ($N \in \mathbb{N} \cup \{\bot\}$)
$\mid A : a\#N$	revealed secret of A ($N \in \mathbb{N}$)
$\mid \Gamma \mid \Gamma'$	parallel composition

Further, $\Gamma \mid t$ is a *timed* configuration, where $t \in \mathbb{N}$ is a global time.

We now illustrate the BitML semantics through a series of examples, which cover all the primitives. When time is immaterial, we will only show the steps of the untimed semantics. A full execution of the timed commitment contract is shown in Appendix A.

Deposits. When a participant A owns a deposit $\langle A, v\rangle_x$, she can employ that amount for several operations: she can divide the deposit into two smaller deposits, or join it with another deposit of hers to form a larger one; the deposit can also be transferred to another participant, or destroyed. For instance, to authorize the join of two deposits, A can perform the following step:

$$\langle A, v\rangle_x \mid \langle A, v'\rangle_y \;\to\; \langle A, v\rangle_x \mid \langle A, v'\rangle_y \mid A[\chi_x]$$

where $\chi_x = x, y \rhd \langle A, v + v'\rangle$ means that A authorizes to spend x. After A also provides the dual authorization χ_y, any participant can perform the actual join as follows:

$$\langle A, v\rangle_x \mid \langle A, v'\rangle_y \mid A[\chi_x] \mid A[\chi_y] \;\to\; \langle A, v + v'\rangle_z$$

Advertisement. Any participant, at any time, can advertise a new contract C (with preconditions G) by performing the following step:

$$\Gamma \;\to\; \Gamma \mid \{G\}C$$

The rule requires that all the deposits mentioned in G exist in Γ, that secrets names are fresh, and that at least one of the participants in G is honest. The last condition, useful to obtain computational soundness, does not limit the power of adversary. Indeed, the same effect of executing a contract among dishonest participants can be obtained by the adversary using the deposit rules, only.

Stipulation. To perform a stipulation, turning a contract advertising into an active contract, a few steps are needed. For instance, consider $G = \mathsf{A} : ! \; 1\!\!\Bbbk \; @ \, x \mid \mathsf{A} : ? \; 1\!\!\Bbbk \; @ \, y \mid \mathsf{A} : \texttt{secret} \, a$, and let C be an arbitrary contract, only involving A. Assuming A honest, our semantics gives the following steps:

$$\begin{aligned} & \langle \mathsf{A}, 1\!\!\Bbbk \rangle_x \mid \langle \mathsf{A}, 1\!\!\Bbbk \rangle_y \mid \{G\}C \\ \to \; & \langle \mathsf{A}, 1\!\!\Bbbk \rangle_x \mid \langle \mathsf{A}, 1\!\!\Bbbk \rangle_y \mid \{G\}C \mid \{\mathsf{A} : a\#N\} \mid \mathsf{A}[\# \rhd \{G\}C] \qquad = \Gamma \\ \to \; & \Gamma \mid \mathsf{A}[x \rhd \{G\}C] \\ \to \; & \langle \mathsf{A}, 1\!\!\Bbbk \rangle_y \mid \langle C, 1\!\!\Bbbk \rangle_z \mid \{\mathsf{A} : a\#N\} \end{aligned}$$

The rules require all participants to commit to their secrets (and to their lengths). Above, this step adds $\{\mathsf{A} : a\#N\} \mid \mathsf{A}[\# \rhd \{G\}C]$ to the configuration, where N is the committed length. After that, all participants must authorize to spend their persistent deposits. Above, this step adds $\mathsf{A}[x \rhd \{G\}C]$ to the configuration. After all such authorizations have been performed, any participant can spend the persistent deposits to create the active contract. Above, this step consumes $\langle \mathsf{A}, 1\!\!\Bbbk \rangle_x$ (and all the authorizations) to create $\langle C, 1\!\!\Bbbk \rangle_z$.

We anticipate that, in the computational setting, committing to a secret a is performed by generating a random nonce s_a and broadcasting its hash $h_a = H(s_a)$. Note that a dishonest A could perform a fake commitment, by broadcasting a value h_a without knowing its preimage. In the symbolic setting, we model this situation by allowing a dishonest A to produce a term $\{\mathsf{A} : a\#\bot\}$, where $\bot$ represents an "invalid" hash. In the subsequent steps, A will not be able to reveal the secret, coherently with the fact that, in the computational setting, A is not able to compute a preimage of h_a.

Withdraw. We now exemplify the rules for active contracts. Executing `withdraw` A transfers the whole contract balance to A:

$$\langle \texttt{withdraw} \; \mathsf{A} + C', v \rangle_x \; \to \; \langle \mathsf{A}, v \rangle_y$$

This step terminates the contract, and creates a deposit owned by A, with a fresh name y. Above, `withdraw` A is executed as a branch within a choice: as usual, taking a branch discards the other ones (denoted as C'). Below, C' always denotes the discarded branches.

Split. `split` can be used to spawn several new concurrent contracts, dividing the balance among them:

$$\langle (\texttt{split} \; v_1 \to C_1 \mid v_2 \to C_2) + C', v_1 + v_2 \rangle_x \to \langle C_1, v_1 \rangle_y \mid \langle C_2, v_2 \rangle_z$$

The balance of the initial contract, $(v_1 + v_2)\!\!\Bbbk$, is split between two newly spawned contracts: C_1, receiving $v_1\!\!\Bbbk$, and C_2, receiving $v_2\!\!\Bbbk$. After this step, the two new contracts are executed independently.

Put. `put` x rakes the volatile deposit x within the contract:

$$\langle \texttt{put} \; x.C + C', v \rangle_y \mid \langle \mathsf{A}, v' \rangle_x \; \to \; \langle C, v + v' \rangle_z$$

This step can only be performed if x is still unspent; otherwise, the `put` prefix is stuck. This action naturally generalizes to multiple volatile deposits $\vec{x}$ (if any of them is spent, the prefix is stuck).

Reveal. `reveal` a `if` p can be fired when the previously committed secret a has been revealed, and it satisfies the guard p. E.g.:

$$\begin{aligned} & \langle \texttt{reveal} \; a \; \texttt{if} \; |a| = M. \, C + C', v \rangle_x \mid \{\mathsf{A} : a\#N\} \\ \to \; & \langle \texttt{reveal} \; a \; \texttt{if} \; |a| = M. \, C + C', v \rangle_x \mid \mathsf{A} : a\#N \\ \to \; & \langle C, v \rangle_y \mid \mathsf{A} : a\#N \qquad\qquad \text{if } N = M \end{aligned}$$

In the first step, A reveals her secret a. In the second step, any participant can cause the contract to take the `reveal` branch, provided that the length of a is M, as required by the predicate. This action naturally generalizes to the case of multiple secrets $\vec{a}$ (all of the must be revealed), and to the case where `put` and `reveal` are performed atomically, e.g. in an action `put` $\vec{x}$ & `reveal` $\vec{a}$ `if` p.

Authorizations. When a branch is decorated by $\mathsf{A} : \cdots$ it can be taken only after A has provided her authorization. For instance:

$$\begin{aligned} & \langle \mathsf{A} : \texttt{withdraw} \; \mathsf{B} + C', v \rangle_x \\ \to \; & \langle \mathsf{A} : \texttt{withdraw} \; \mathsf{B} + C', v \rangle_x \mid \mathsf{A}[x \rhd \mathsf{A} : \texttt{withdraw} \; \mathsf{B}] \to \langle \mathsf{B}, v \rangle_y \end{aligned}$$

In the first step, A authorizes the contract to take the branch `withdraw` B. After that, any participant can fire such branch. When multiple authorizations are required, the branch can be taken only after all of them occur in the configuration.

Time. We now discuss the rules for handling time. These rules describe transitions between timed configurations $\Gamma \mid t$ where t denotes the current time. We always allow time t to advance by a delay $\delta > 0$, through the rule: $\Gamma \mid t \to \Gamma \mid t + \delta$. We allow a contract branch decorated with `after` t to be taken only when the current time is greater than t. For instance, if $t_0 + \delta \geq t$:

$$\begin{aligned} & \langle \texttt{after} \; t : \texttt{withdraw} \; \mathsf{B}, v \rangle_x \mid t_0 \\ \to \; & \langle \texttt{after} \; t : \texttt{withdraw} \; \mathsf{B}, v \rangle_x \mid t_0 + \delta \; \to \; \langle \mathsf{B}, v \rangle_y \mid t_0 + \delta \end{aligned}$$

For the branches not guarded by an `after`, we lift transitions from untimed to timed configurations, without making them affect time. Namely, for an untimed transition $\Gamma \to \Gamma'$, we also have the timed transition $\Gamma \mid t \to \Gamma' \mid t$. This makes actions instantaneous (similarly to many timed process calculi [50]) reflecting the assumption that participants can always meet deadlines, if they want to.

5 FURTHER BITML EXAMPLES

We show a few additional examples of BitML contracts.

Variable-refund escrow. We propose a variant of the escrow contract in Section 2 where, similarly to [2, 25], M can issue a *partial* refund of $\zeta\!\!\Bbbk$ to A, and of $(1-\zeta)\!\!\Bbbk$ to B. The possible values of ζ are given by a finite set Z in the range $[0, 1]$. We model the new contract as follows:

$$\begin{aligned} \mathit{VarEscrow} &= \mathsf{A} : \texttt{withdraw} \; \mathsf{B} + \mathsf{B} : \texttt{withdraw} \; \mathsf{A} + \textstyle\sum_{\zeta \in Z} \mathsf{M} : D_\zeta \\ D_\zeta &= \texttt{split} \left(\zeta \to \texttt{withdraw} \; \mathsf{A} \mid (1-\zeta) \to \texttt{withdraw} \; \mathsf{B} \right) \end{aligned}$$

The case of full refunds is obtained with $Z = \{0, 1\}$. If $Z = \{0, 1/2, 1\}$, M can also choose to refund $1/2\!\!\Bbbk$ to both.

Mutual timed commitment. We can also model *mutual* timed commitment as follows, where $t < t'$:

$$\begin{aligned} G &= \mathsf{A} : ! \; 1\!\!\Bbbk \; @ \, x \mid \mathsf{A} : \texttt{secret} \; a \mid \mathsf{B} : ! \; 1\!\!\Bbbk \; @ \, y \mid \mathsf{B} : \texttt{secret} \; b \\ \mathit{MTC} &= \texttt{reveal} \; a. \, C \; + \; (\texttt{after} \; t : \texttt{withdraw} \; \mathsf{B}) \\ C &= \texttt{reveal} \; b. \, C' \; + \; (\texttt{after} \; t' : \texttt{withdraw} \; \mathsf{A}) \\ C' &= \texttt{split} \; (1\!\!\Bbbk \to \texttt{withdraw} \; \mathsf{A} \mid 1\!\!\Bbbk \to \texttt{withdraw} \; \mathsf{B}) \end{aligned}$$

The contract MTC can reduce to C if A reveals a, otherwise (after t), B can redeem $2\!\!\Bbbk$. In C, if B reveals b, then both participants can redeem their deposits, running C'. Otherwise, A can redeem $2\!\!\Bbbk$.

Zero-collateral lottery. We show a two-players lottery which requires no collateral, similarly to [19, 47]. The preconditions just require each participant to bet 1₿, and to commit to a secret:

$$\mathsf{A} : !\ 1฿\ @\ x \ \mid\ \mathsf{A} : \mathtt{secret}\ a \ \mid\ \mathsf{B} : !\ 1฿\ @\ y \ \mid\ \mathsf{B} : \mathtt{secret}\ b$$

The contract is the following, where $t < t'$:

$$\begin{aligned}&\mathtt{reveal}\ b\ \mathtt{if}\ 0 \le |b| \le 1.\big(\ \mathtt{reveal}\ ab\ \mathtt{if}\ |a| = |b|.\ \mathtt{withdraw}\ \mathsf{A}\\ &\qquad\qquad + \ \mathtt{reveal}\ ab\ \mathtt{if}\ |a| \neq |b|.\ \mathtt{withdraw}\ \mathsf{B}\\ &\qquad\qquad + \ \mathtt{after}\ t' : \mathtt{withdraw}\ \mathsf{B}\big)\\ &+\ \mathtt{after}\ t : \mathtt{withdraw}\ \mathsf{A}\end{aligned}$$

Here, B must reveal first. If B does not reveal his secret by the deadline t, or the secret has not the expected length, then A can redeem 2₿. Otherwise, A in turn must reveal by the deadline t', or let B redeem 2₿. If both A and B reveal, then the winner is determined by comparing the lengths of their secrets. As before, the rational strategy for each player is to choose a secret length 0 or 1, and reveal it. This makes the lottery fair, even in the absence of a collateral.

Rock-Paper-Scissors. Using similar insights, we can craft contracts for other games. For instance, consider *Rock-Paper-Scissors*, a two-players hand game where both players choose simultaneously a hand-shape, and the winner is decided along with the following rules: rock beats scissors, scissors beats paper, and paper beats rock. We model the game for two players A and B who bet 1₿ each, and represent their moves as secrets of length 0 (rock), 1 (paper), and 2 (scissors). The boolean predicate $w(N, M)$ is true when the move N wins over M:

$$w(N, M) \ = \ (N = 0 \wedge M = 2) \vee (N = 2 \wedge M = 1) \vee (N = 1 \wedge M = 0)$$

The contract preconditions are:

$$\mathsf{A} : !\ 3฿\ @\ x \mid \mathsf{A} : \mathtt{secret}\ a \mid \mathsf{B} : !\ 3฿\ @\ y \mid \mathsf{B} : \mathtt{secret}\ b$$

while the contract is the following:

$$\begin{aligned}&\mathtt{split}\big(\\ &\ 2฿ \to \mathtt{reveal}\ b\ \mathtt{if}\ 0 \le |b| \le 2.\ \mathtt{withdraw}\ \mathsf{B}\\ &\qquad + \mathtt{after}\ t : \mathtt{withdraw}\ \mathsf{A}\\ &\mid 2฿ \to \mathtt{reveal}\ a\ \mathtt{if}\ 0 \le |a| \le 2.\ \mathtt{withdraw}\ \mathsf{A}\\ &\qquad + \mathtt{after}\ t : \mathtt{withdraw}\ \mathsf{B}\\ &\mid 2฿ \to \mathtt{reveal}\ ab\ \mathtt{if}\ w(|a|, |b|).\ \mathtt{withdraw}\ \mathsf{A}\\ &\qquad + \mathtt{reveal}\ ab\ \mathtt{if}\ w(|b|, |a|).\ \mathtt{withdraw}\ \mathsf{B}\\ &\qquad + \mathtt{reveal}\ ab\ \mathtt{if}\ |a| = |b|.\ \mathtt{split}(1฿ \to \mathtt{withdraw}\ \mathsf{A}\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mid 1฿ \to \mathtt{withdraw}\ \mathsf{B})\big)\end{aligned}$$

The contract is split in three parts, each with a balance of 2₿: the first two parts allow the players to redeem the collaterals by revealing their secrets in time, while the third one computes the winner. The winner is A if $w(|a|, |b|)$, and B if $w(|b|, |a|)$. If a and b have the same length (i.e., they represent the same move), then there is a tie, so the bets are given back to the two players. Notice that if a player chooses a secret of unexpected length, then it may happen that the 2₿ in the third part of the `split` remain frozen. However, in such case the dishonest player will pay a 2₿ penalty to the other one. A zero-collateral version of *Rock-Paper-Scissors* can be obtained similarly to the zero-collateral lottery.

6 SYMBOLIC STRATEGIES & ADVERSARIES

Symbolic runs. A *symbolic run* R^s is a (possibly infinite) sequence $\Gamma_0 \mid t_0 \xrightarrow{\alpha_0} \Gamma_1 \mid t_1 \xrightarrow{\alpha_1} \cdots$ where α_i are the transition labels, Γ_0 contains only deposits, and $t_0 = 0$. If R^s is finite, we write Γ_{R^s} for its last untimed configuration.

Stripping. The strategy of a participant can inspect the whole past run, except for the (lengths of the) unrevealed secrets. The *stripping* of a run censors this information: technically, it replaces each committed secret $\{\mathsf{A} : a\#N\}$ with a term $\{\mathsf{A} : a\#\bot\}$.

Symbolic participant strategies. A symbolic strategy Σ^s_{A} is a PPTIME algorithm which allows A to select which action(s) to perform, among those permitted by the BitML semantics. Σ^s_{A} receives as input a stripped run R^s_*, and outputs a finite set of actions (possibly, time delays) that A wants to perform. The choice among these actions is controlled by the adversary strategy, specified below. We forbid Σ^s_{A} to output authorizations for participants $\mathsf{B} \neq \mathsf{A}$. Further, strategies must be *persistent*: if on a run Σ^s_{A} chooses an action α, and α is not taken as the next step in the run (e.g., because some other participant acts earlier), then Σ^s_{A} must still choose α after that step, if still enabled. In this way, once Σ^s_{A} has chosen α (implicitly, giving to the adversary her consent to schedule such action), she cannot change her mind.

Symbolic adversary strategies. The adversary Adv acts on behalf of all the dishonest participants, and controls the scheduling among all participants (including the honest ones). Her symbolic strategy Σ^s_{Adv} takes as input the current run and the sets of moves outputted by the strategies of honest participants. Both the run and the moves are stripped, to prevent the adversary from inferring the lengths of secrets. The output of Σ^s_{Adv} is a single action λ^s (to be appended to the current run), only subject to the following constraints: (i) if λ^s is an authorization by some honest A, then it must be chosen by Σ^s_{A}; (ii) if λ^s is a time delay, then all the honest participants must agree. Condition (i) rules out authorization forgeries, while (ii) prevents the adversary from delaying the honest participants. Were this condition dropped, honest participants could be prevented from meeting deadlines: e.g., in the timed commitment contract, the adversary could collude with B by enabling the `after` branch before A had the possibility to reveal her secret.

Symbolic conformance. Strategies are PPTIME algorithms: so, besides the other inputs, they also implicitly take as input a stream r of random bits. Fixing this stream and a set of strategies Σ^s — both for the honest participants and for the adversary — we obtain a unique run, which is made by the sequence of actions chosen by Σ^s_{Adv} when taking as input the outputs of the honest participants' strategies. We say that this run is *conformant* to Σ^s (and r). When $\Sigma^s_{\mathsf{Adv}} \notin \Sigma^s$, we say that R^s conforms to Σ^s (and r) when there exists some Σ^s_{Adv} such that R^s conforms to $\Sigma^s \cup \{\Sigma^s_{\mathsf{Adv}}\}$ (and r).

7 COMPUTATIONAL MODEL

In this section we introduce our computational model, which will be the target of the BitML compiler. We start by briefly recapping Bitcoin transactions (referring to [15] for a full formal model).

Transactions. In Bitcoin, *transactions* describe transfers of currency. The log of all transactions is maintained on a public, immutable and decentralised data structure called *blockchain*. We represent transactions as records, with fields in, wit, out and absLock. For instance, consider the transactions T_1 and T_2 below:

T_1	
in:	$\cdots$
wit:	$\cdots$
out:	$0 \mapsto (\lambda\varsigma.\mathrm{versig}_{\mathsf{K(A)}}(\varsigma), v_0\text{฿})$
	$1 \mapsto (\lambda\varsigma.\mathrm{versig}_{\mathsf{K(B)}}(\varsigma), v_1\text{฿})$

T_2	
in:	$0 \mapsto (\mathsf{T}_1, 0)$
wit:	$0 \mapsto sig_{\mathsf{K(A)}}$
out:	$0 \mapsto (\lambda x.H(x) = k, v_0\text{฿})$
absLock:	t

The transaction T_1 has two *outputs*: the v_0฿ in out(0) can be redeemed by any transaction T whose in field refers to $(\mathsf{T}_1, 0)$, and whose wit field satisfies the predicate in out(0) (similarly for the other output). This is the case e.g. of the transaction T_2 above. Its *witness* $sig_{\mathsf{K(A)}}$ is the signature of A on T_2 (excluding the wit field itself), as required by T_1.out(0).

If T_1 is on the blockchain and its out(0) is unspent, A can update the blockchain by appending T_2. This moves v_0฿[1] from T_1 to T_2. The transaction T_2 has only one output, which can be redeemed by any transaction providing a witness having hash k. The time t in T_2.absLock represents the earliest moment when T_2 can be put on the blockchain. A subsequent transaction can redeem $(v_1 + v_0)$฿ in a single shot. This requires two inputs, $(\mathsf{T}_1, 1)$ and $(\mathsf{T}_2, 0)$, and two witnesses. The witness associated to the first input is a signature of B; the other is a preimage of k.

We assume that Bitcoin uses secure cryptographic primitives, i.e. ideal hash functions (which hereafter are modelled according to the random oracle model [21]), and a digital signature scheme which is robust against existential forgery attacks.

Blockchain. A blockchain $\mathbf{B}$ is a sequence $(\mathsf{T}_0, t_0) \cdots (\mathsf{T}_n, t_n)$, where $\mathsf{T}_0 \cdots \mathsf{T}_n$ are transactions (T_0 is the *coinbase* transaction, i.e. T_0.in $= \bot$, meaning that is does not point to a previous transaction), and $t_0 \cdots t_n$ are timestamps, with $t_i \leq t_j$ for all $i \leq j$. Hereafter, we assume that blockchains are append-only, without forks, and *consistent*, i.e. obtained by appending transactions that respect the Bitcoin protocol (as formalised in [15]). Essentially, these ideal properties are coherent with the results in [16, 34, 40], as long as the adversary does not control a very large mining pool [12, 33].

Computational runs. We now introduce the computational counterparts of symbolic runs. These are sequences of *computational labels* λ^c, namely bitstrings encoding one of the following actions, where $\mathsf{A} \in \mathsf{Part} \cup \{\mathsf{Adv}\}$, and m is a bitstring:

$\mathsf{A} \rightarrow * : m$	A broadcasts message m
T	append a Bitcoin transaction to the blockchain
δ	perform a delay

To compute the hash of a message m, A sends m to the oracle $\mathsf{O} \notin \mathsf{Part}$, and waits for the reply $H(m)$. Also these actions are included in the computational labels. Note that reliable message broadcasts can be effectively obtained through the Bitcoin network. We postulate that a computational run begins with a coinbase transaction T_0, followed by the broadcasts of the public keys of all participants. Each A has two key pairs: K_A for signing messages, and $\hat{K}_\mathsf{A}$ for redeeming deposits. For each A (either honest or not), we assume that T_0 has an output redeemable with the private key $\hat{K}^s_\mathsf{A}$. In this way, each participant starts with some funds (possibly 0), and knows the public keys of the others. By extracting the transactions from a computational run R^c, and assigning their time according to the accumulated delays, we obtain a blockchain, denoted as $\mathbf{B}_{R^c}$.

Stripping. Analogously to the stripping of symbolic runs, we define the A-*stripping* of a computational run R^c as the run obtained by removing from R^c all the messages *not* visible by A, i.e. the messages between O and some other $\mathsf{B} \neq \mathsf{A}$.

Computational participant strategies. We now introduce the computational counterparts of symbolic strategies. A computational strategy Σ^c_A for A is a PPTIME algorithm which receives as input a (A-stripped) computational run R^c_*, and outputs is a finite set of computational labels. The choice among these labels is controlled by the adversary strategy, specified below. We impose a few sanity constraints: (i) we forbid A to impersonate another participant; (ii) if the strategy outputs a transaction T, then T must be a consistent update of the blockchain $\mathbf{B}_{R^c_*}$ obtained from the run in input, and all the witnesses of T have already been broadcast in the run; (iii) finally, strategies must be persistent, similarly to the symbolic case. The condition on the witnesses of T in (ii) is not a limitation, since these witnesses become public once T is broadcast on the Bitcoin network. We put this condition since it helps to obtain a sharp correspondence between computational and symbolic runs: indeed, the symbolic counterpart of broadcasting a witness is to give an authorization.

Computational adversary strategies. A computational adversary strategy Σ^c_{Adv} is a PPTIME algorithm taking as input a (Adv-stripped) computational run R^c_* and the moves chosen by each honest participant. The strategy gives as output a single computational label, to be appended to the run. We require Σ^c_{Adv} to obey sanity constraints similar to those of participants strategies, with two differences: (i) Adv can impersonate any other participant (except the oracle O); (ii) Adv can perform a time delay only if all the honest participants agree.

This assumption prevents the adversary from delaying the actions of honest participants, who can therefore always meet their deadlines, without interference by the adversary. Note that, even if the adversary controlled a large portion of the mining power of the Bitcoin network — so being able to delay the transactions sent by honest participants — honest participants could still protect themselves by setting far enough deadlines in their contracts, making the delay attacks ineffective[2].

Computational conformance. For each participant A, we indicate with r_A the stream of random bits implicitly used by Σ^c_A. Similarly, we indicate with r_O the stream available to the oracle. Fixing a set of computational strategies Σ^c — both for the honest participants and for the adversary — and a function r from participants to streams, we obtain a unique computational run, made by

[1] In the actual Bitcoin, the value the outputs of T_2 must be strictly smaller than v_0, and the difference is paid to the Bitcoin network. For simplicity, in this paper (as in [15]) we neglect these *transaction fees*.

[2] For the sake of simplicity, we do not model such delays. At the price of adding further complexity, we could extend our model to relax this assumption, by allowing the adversary to perform delays under reasonable constraints. As long as the same delays are allowed in both the symbolic and computational models, computational soundness still holds. In such extended model, deadlines in contracts must be chosen so to compensate for these delays.

the sequence of actions chosen by Σ^c_{Adv} when taking as input the outputs of the honest participants' strategies, and enforcing that the queries to the oracle are always answered. We say that this run is *conformant* to Σ^c and r.

8 COMPILING BITML TO BITCOIN

We now describe how to implement BitML on top of Bitcoin. This is done by *compiling* BitML contract advertisements $\{G\}C$ into a finite set $\mathcal{T}$ of Bitcoin transactions[3], signed by all participants (except from those whose authorizations are required after stipulation, via $\mathsf{A} : \cdots$). In Bitcoin, appending (a subset of) these transactions to the blockchain mimics, in BitML, the semantics of $\{G\}C$ (assuming, as usual, that at least one participant is honest). More precisely, the first transaction in $\mathcal{T}$ to be published (called T_{init}), redeems all the permanent deposits, correspondingly to the stipulation of $\{G\}C$. After that, T_{init} can only be redeemed by one of the transactions in $\mathcal{T}$ which corresponds to the subsequent computation step of $\langle C, v\rangle$ in BitML. This is enforced by requiring, in the output script of T_{init}, suitable signatures by all the participants involved in $\{G\}C$. The same principle is followed for the other transactions in $\mathcal{T}$: they can be appended to the blockchain only when this corresponds to a computation step in BitML. The only transactions in $\mathcal{T}$ that can be redeemed by transactions *not* in $\mathcal{T}$ are those which correspond to the deposits $\langle \mathsf{A}, v\rangle$ obtained by reducing a $\texttt{withdraw}\ \mathsf{A}$ contract. Indeed, these transactions can be redeemed by standard transactions signed by A.

We illustrate the compiler through a series of examples, which cover all the primitives of BitML. We refer to Appendix A and [20] for a formal definition.

Before running the compiler, participants generate the following key pairs, and exchange their public parts:

- $\mathsf{K}(\mathsf{A})$, used in the compilation of a $\texttt{withdraw}\ \mathsf{A}$ contract. We exploit this to guarantee that the deposit obtained after the execution of $\texttt{withdraw}\ \mathsf{A}$ can be redeemed with a signature of A (using the private part of $\mathsf{K}(\mathsf{A})$).
- $\mathsf{K}(D, \mathsf{A})$ (for each subterm D of C), used in the compilation of the subterms of C of the form $D + C'$ to enable the firing of the (initial) action in the D branch. The private part of $\mathsf{K}(D, \mathsf{A})$ is used to compute the witness of the transaction corresponding to the compilation of D.

For a set of participants $\mathcal{P} = \{\mathsf{A}_1, \ldots, \mathsf{A}_n\}$, we denote with $\mathsf{K}(D, \mathcal{P})$ the set of key pairs $\{\mathsf{K}(D, \mathsf{A}_1), \ldots, \mathsf{K}(D, \mathsf{A}_n)\}$.

Withdraw. As a first example, we show how to compile:

$$\{G\}C \;=\; \{\mathsf{A} : !\ 1\text{฿}\ @\ x \mid \mathsf{B} : !\ 1\text{฿}\ @\ y\}\ \texttt{withdraw}\ \mathsf{B}$$

The transactions obtained from the compiler are the following, where T_x is the transaction associated to A's deposit $\langle \mathsf{A}, 1\text{฿}\rangle_x$ and T_y is the transaction associated to B's deposit $\langle \mathsf{B}, 1\text{฿}\rangle_y$:

T_{init}	T_B
in: $0 \mapsto \mathsf{T}_x, 1 \mapsto \mathsf{T}_y$	in: T_{init}
wit: $0 \mapsto \mathsf{sig}_{\mathsf{K}(\mathsf{A})}, 1 \mapsto \mathsf{sig}_{\mathsf{K}(\mathsf{B})}$	wit: $\mathsf{sig}_{\mathsf{K}(\texttt{withdraw}\ \mathsf{B}, \{\mathsf{A},\mathsf{B}\})}$
out: $(\lambda\vec{\varsigma}.\ \mathsf{versig}_{\mathsf{K}(\texttt{withdraw}\ \mathsf{B}, \{\mathsf{A},\mathsf{B}\})}(\vec{\varsigma}),\ 2\text{฿})$	out: $(\lambda\varsigma.\ \mathsf{versig}_{\mathsf{K}(\mathsf{B})}(\varsigma),\ 2\text{฿})$

In BitML, the stipulation of the contract requires a few steps:

$$\begin{aligned}&\langle \mathsf{A}, 1\text{฿}\rangle_x \mid \langle \mathsf{B}, 1\text{฿}\rangle_y \mid \{G\}C\\ \to^4 &\langle \mathsf{A}, 1\text{฿}\rangle_x \mid \langle \mathsf{B}, 1\text{฿}\rangle_y \mid \{G\}C \mid \mathsf{A}[x \rhd \{G\}C] \mid \mathsf{B}[y \rhd \{G\}C]\\ &\qquad \mid \mathsf{A}[\# \rhd \{G\}C] \mid \mathsf{B}[\# \rhd \{G\}C] \;=\; \Gamma\end{aligned}$$

In Bitcoin, these steps correspond to obtaining the transactions above from the compiler, and exchanging the signatures shown there (signing T_{init} *last*). The BitML authorization $\mathsf{A}[x \rhd \{G\}C]$ corresponds to the broadcast of A'signature in the first witness of T_{init}. The authorization $\mathsf{A}[\# \rhd \{G\}C]$ in this case is immaterial: in general, it corresponds to the commit of A's secrets. The authorizations $\mathsf{B}[y \rhd \{G\}C]$ and $\mathsf{B}[\# \rhd \{G\}C]$ are similar, for B. In BitML, the stipulation is then completed with the following step:

$$\Gamma \;\to\; \langle \texttt{withdraw}\ \mathsf{B}, 2\text{฿}\rangle$$

In Bitcoin, this corresponds to appending T_{init} to the blockchain, which redeems T_x and T_y. The last computation step in BitML is:

$$\langle \texttt{withdraw}\ \mathsf{B}, 2\text{฿}\rangle \;\to\; \langle \mathsf{B}, 2\text{฿}\rangle_z$$

In Bitcoin, this corresponds to appending T_B to the blockchain, which redeems 2฿ from T_{init}[4]. After that, 2฿ are under B's control, since the output script of T_B only requires B's signature. The unspent transaction T_B corresponds to the BitML deposit $\langle \mathsf{B}, 2\text{฿}\rangle_z$.

Note that, during the stipulation phase, it is crucial to exchange the signatures on T_{init} *after* the ones on T_B, otherwise we would lose the correspondence with the BitML semantics. Indeed, were T_{init} signed first, A could refuse to sign T_B, and yet be able to complete the stipulation, by appending T_{init} to the blockchain. Then, B would be prevented to append T_B, i.e. the Bitcoin counterpart of the BitML $\texttt{withdraw}\ \mathsf{B}$ action, which would transfer 2฿ to him.

Authorizations. We exploit the previous example to illustrate the compilation of authorizations. Compiling $\mathsf{A} : \texttt{withdraw}\ \mathsf{B}$ requires only minor changes: in the witness of T_B, the compiler only inserts B's signature $\mathsf{sig}_{\mathsf{K}(\mathsf{A}\ :\ \texttt{withdraw}\ \mathsf{B}, \mathsf{B})}$, while the output script of T_{init} still requires both signatures, so to prevent T_B from being appended the blockchain before obtaining A's authorization. In BitML, to perform the $\texttt{withdraw}$ action, A must authorize it:

$$\Gamma \;\to\; \Gamma \mid \mathsf{A}[\mathsf{A} : \texttt{withdraw}\ \mathsf{B}]$$

In Bitcoin, this step corresponds to broadcasting A's signature $\mathsf{sig}_{\mathsf{K}(\mathsf{A}\ :\ \texttt{withdraw}\ \mathsf{B}, \mathsf{A})}$. After that, the signature can be added to the witness of T_B, which can now be appended to the blockchain.

[3] An implementation of the BitML compiler is under way, at https://github.com/bitml-lang/bitml-compiler. This implementation generates standard Bitcoin transactions by exploiting our BALZaC tool [7]. This is crucial, since the Bitcoin network currently discards non-standard transactions.

[4] Note that in the example above, an optimized version of our compiler could omit the transaction T_{init}, and transfer the deposit to B directly through T_B (with the in field as in T_{init}). However, this optimization is only possible when $\texttt{withdraw}$ does not occur within a choice; further, the semantics of $\texttt{withdraw}$ should be revised accordingly to preserve computational soundness. For the sake of simplicity, we prefer to ignore this optimization in the definition of our compiler.

After. Similarly, compiling `after` t : `withdraw` B requires only a small change to our first example: the transaction T_B now has an absLock field set to time t. In this way, T_B can not be appended to the blockchain until time t — coherently to the BitML semantics, where the `withdraw` B action can not be fired until such time.

Split. To illustrate the compilation of the `split` primitive, let:

$$C_s \;=\; \texttt{split}\ (1฿ \to \texttt{withdraw}\ \mathsf{A} \mid 2฿ \to \texttt{withdraw}\ \mathsf{B})$$

Compiling $\{\mathsf{A} : !\ 2฿\,@\,x \mid \mathsf{B} : !\ 1฿\,@\,y\}\,C_s$ produces the following transactions, where T_x and T_y correspond to the deposits $\langle \mathsf{A}, 2฿\rangle_x$ and $\langle \mathsf{B}, 1฿\rangle_y$, and $KW^p = \mathsf{K}(\texttt{withdraw}\ \mathsf{p}, \{\mathsf{A},\mathsf{B}\})$ for $p \in \{\mathsf{A},\mathsf{B}\}$:

T_{init}	T_{split}
in: $0 \mapsto \mathsf{T}_x, 1 \mapsto \mathsf{T}_y$ wit: $0 \mapsto \mathrm{sig}_{\mathsf{K}(\mathsf{A})}, 1 \mapsto \mathrm{sig}_{\mathsf{K}(\mathsf{B})}$ out: $(\lambda\vec{\varsigma}.\mathrm{versig}_{\mathsf{K}(C_s,\{\mathsf{A},\mathsf{B}\})}(\vec{\varsigma}), 3฿)$	in: T_{init} wit: $\mathrm{sig}_{\mathsf{K}(C_s,\{\mathsf{A},\mathsf{B}\})}$ out: $0 \mapsto (\lambda\vec{\varsigma}.\mathrm{versig}_{KW^{\mathsf{A}}}(\vec{\varsigma}), 1฿)$ $1 \mapsto (\lambda\vec{\varsigma}.\mathrm{versig}_{KW^{\mathsf{B}}}(\vec{\varsigma}), 2฿)$

T_{A}	T_{B}
in: $(\mathsf{T}_{split}, 0)$ wit: $\mathrm{sig}_{\mathsf{K}(\texttt{withdraw}\ \mathsf{A},\{\mathsf{A},\mathsf{B}\})}$ out: $(\lambda\varsigma.\ \mathrm{versig}_{\mathsf{K}(\mathsf{A})}(\varsigma),\ 1฿)$	in: $(\mathsf{T}_{split}, 1)$ wit: $\mathrm{sig}_{\mathsf{K}(\texttt{withdraw}\ \mathsf{B},\{\mathsf{A},\mathsf{B}\})}$ out: $(\lambda\varsigma.\ \mathrm{versig}_{\mathsf{K}(\mathsf{B})}(\varsigma),\ 2฿)$

As before, T_{init} gathers A's and B's deposits and starts the contract. Then, appending T_{split} to the blockchain splits the contract balance between two different outputs, indexed with 0 and 1. In BitML, this would correspond to the computation step:

$$\langle C_s, 3฿\rangle \;\to\; \langle \texttt{withdraw}\ \mathsf{A}, 1฿\rangle \mid \langle \texttt{withdraw}\ \mathsf{B}, 2฿\rangle$$

where the two contracts in the parallel composition can be executed independently (as usual in process calculi). Similarly, the two outputs of T_{split} can be independently redeemed by T_{A} and T_{B}. The in field of these transactions specifies, besides the input transaction T_{split}, also the index of the output they want to redeem. For instance, appending T_{A} corresponds, in BitML, to the step:

$$\langle \texttt{withdraw}\ \mathsf{A}, 1฿\rangle \mid \langle \texttt{withdraw}\ \mathsf{B}, 2฿\rangle \to \langle \mathsf{A}, 1฿\rangle \mid \langle \texttt{withdraw}\ \mathsf{B}, 2฿\rangle$$

Put. To illustrate the compilation of the `put` primitive, let:

$$C_p \;=\; \texttt{put}\ x.\ \texttt{withdraw}\ \mathsf{B}$$

Compiling $\{\mathsf{A} : ?\ 1฿\,@\,x \mid \mathsf{A} : !\ 1฿\,@\,y \mid \mathsf{B} : !\ 1฿\,@\,z\}\,C_p$ produces the following transactions, where $\mathsf{T}_x, \mathsf{T}_y, \mathsf{T}_z$ are the Bitcoin counterpart of the BitML deposits:

T_{init}	T_{B}
in: $0 \mapsto \mathsf{T}_y, 1 \mapsto \mathsf{T}_z$ wit: $0 \mapsto \mathrm{sig}_{\mathsf{K}(\mathsf{A})}, 1 \mapsto \mathrm{sig}_{\mathsf{K}(\mathsf{B})}$ out: $(\lambda\vec{\varsigma}.\mathrm{versig}_{\mathsf{K}(C_p,\{\mathsf{A},\mathsf{B}\})}(\vec{\varsigma}), 2฿)$	in: T_{put} wit: $\mathrm{sig}_{\mathsf{K}(\texttt{withdraw}\ \mathsf{B},\{\mathsf{A},\mathsf{B}\})}$ out: $(\lambda\varsigma.\ \mathrm{versig}_{\mathsf{K}(\mathsf{B})}(\varsigma),\ 3฿)$

T_{put}
in: $0 \mapsto \mathsf{T}_{init}, 1 \mapsto \mathsf{T}_x$ wit: $0 \mapsto \mathrm{sig}_{\mathsf{K}(C_p,\{\mathsf{A},\mathsf{B}\})}, 1 \mapsto \mathrm{sig}_{\mathsf{K}(\mathsf{A})}$ out: $(\lambda\vec{\varsigma}.\mathrm{versig}_{\mathsf{K}(\texttt{withdraw}\ \mathsf{B},\{\mathsf{A},\mathsf{B}\})}(\vec{\varsigma}),\ 3฿)$

The transaction T_{init} gathers *only* the persistent deposits. T_{put} has two inputs: T_{init}, which can be redeemed with the signatures of both A and B, and T_x, which can be redeemed with A's signature only. All these signatures are exchanged before stipulation, hence any participant can append T_{put} to the blockchain — provided that T_x is still unspent. Instead, if T_x has been spent, the contract gets stuck, and the deposit within T_{init} is frozen — coherently with the semantics of BitML, where deposits can be spent arbitrarily.

Reveal / choice. Recall from Section 2 the timed commitment contract $TC = D_1 + D_2$, where $D_1 = \texttt{reveal}\ a.\ \texttt{withdraw}\ \mathsf{A}$ and $D_2 = \texttt{after}\ t : \texttt{withdraw}\ \mathsf{B}$. The contract precondition is $G = \mathsf{A} : !\ 1฿\,@\,x \mid \mathsf{A} : \texttt{secret}\ a \mid \mathsf{B} : !\ 0฿\,@\,y$. Before running the compiler, A generates a random nonce s_a, and broadcasts its hash $h_a = H(s_a)$. Honest participants will choose a nonce of length greater than a public security parameter η. A large enough parameter (e.g., $\eta = 128$) ensures that the other participants cannot infer s_a (assuming H to be preimage resistant), nor its length. Further, A cannot later on reveal a different secret or a different length (assuming collision resistance). After that, both participants can compile $\{G\}TC$, obtaining the following transactions:

T_{init}
in: $0 \mapsto \mathsf{T}_x, 1 \mapsto \mathsf{T}_y$ wit: $0 \mapsto \mathrm{sig}_{\mathsf{K}(\mathsf{A})}, 1 \mapsto \mathrm{sig}_{\mathsf{K}(\mathsf{B})}$ out: $(\lambda\vec{\varsigma}\beta.\mathrm{versig}_{\mathsf{K}(D_1,\{\mathsf{A},\mathsf{B}\})}(\vec{\varsigma}) \wedge H(\beta) = h_a \wedge \lvert\beta\rvert \geq \eta$ $\vee\ \mathrm{versig}_{\mathsf{K}(D_2,\{\mathsf{A},\mathsf{B}\})}(\vec{\varsigma}),\ 1฿)$

T_{reveal}
in: T_{init} wit: $\mathrm{sig}_{\mathsf{K}(D_1,\{\mathsf{A},\mathsf{B}\})}\ [s_a]$ out: $(\lambda\vec{\varsigma}.\mathrm{versig}_{\mathsf{K}(\texttt{withdraw}\ \mathsf{A},\{\mathsf{A},\mathsf{B}\})}(\vec{\varsigma}),\ 1฿)$

T_{A}	T_{B}
in: T_{reveal} wit: $\mathrm{sig}_{\mathsf{K}(\texttt{withdraw}\ \mathsf{A},\{\mathsf{A},\mathsf{B}\})}$ out: $(\lambda\varsigma.\ \mathrm{versig}_{\mathsf{K}(\mathsf{A})}(\varsigma),\ 1฿)$	in: T_{init} wit: $\mathrm{sig}_{\mathsf{K}(D_2,\{\mathsf{A},\mathsf{B}\})}$ – out: $(\lambda\varsigma.\ \mathrm{versig}_{\mathsf{K}(\mathsf{B})}(\varsigma),\ 1฿)$ absLock: t

Transaction T_{init} can be redeemed in two ways, according to the two clauses in the disjunction of its output script: (i) with both signatures $\mathrm{sig}_{\mathsf{K}(D_1,\mathsf{A})}$ and $\mathrm{sig}_{\mathsf{K}(D_1,\mathsf{B})}$ (corresponding to the `reveal` branch), or (ii) with both signatures $\mathrm{sig}_{\mathsf{K}(D_2,\mathsf{A})}$ and $\mathrm{sig}_{\mathsf{K}(D_2,\mathsf{B})}$ (corresponding to the `after` branch).

In case (i), also the secret value s_a must be provided in the wit field of T_{reveal}. As indicated by the square brackets around s_a in T_{reveal}, such value is *not* provided at compilation time, but added at runtime. Crucially, altering the wit field does not invalidate the signatures $\mathrm{sig}_{\mathsf{K}(D_1,\mathsf{A})}$ and $\mathrm{sig}_{\mathsf{K}(D_1,\mathsf{B})}$ on T_{reveal} (since signatures neglect the wit field), nor the actual identifier of T_{reveal} used in the in field of T_{A}. This relies on the *SegWit* feature (activated in August 2017), which allows to neglect witnesses in the computation of transaction identifiers. Revealing the secret and appending T_{reveal} correspond to the following computation steps in BitML (time is omitted, because immaterial in this case):

$$\{\mathsf{A} : a\#N\} \mid \langle TC, 1฿\rangle \;\to\; \mathsf{A} : a\#N \mid \langle TC, 1฿\rangle \;\to\; \langle \texttt{withdraw}\ \mathsf{A}, 1฿\rangle$$

After that, anyone can append the transaction T_{A} to the blockchain to transfer 1฿ under A's control. Note that once T_{reveal} is on the blockchain, it will be no longer possible to append T_{B}, since both transactions want to redeem T_{init}.

In case (ii), the absLock guarantees that T_{B} can be appended to the blockchain only after time t, coherently with the `after` clause

in BitML. Indeed, appending T_{B} (which makes 1Ƀ available to B) corresponds to the following step in BitML (where $t' \geq t$):

$$\langle TC, 1Ƀ\rangle \mid t' \rightarrow \langle \mathsf{B}, 1Ƀ\rangle \mid t'$$

9 COHERENCE

Our computational soundness result is based on a correspondence between symbolic and computational runs, that we call *coherence*. Intuitively, a symbolic run R^s is coherent with a computational run R^c when each symbolic step in R^s is matched by the computational step corresponding to its implementation in R^c (in symbols, $R^s \sim_r R^c$, where r is the randomness source used by participants). We illustrate the coherence relation following the possible symbolic steps (see [20] for the formal details).

Advertisement. Advertising a contract $\{G\}C$ in R^s is performed through the following step:

$$\Gamma \rightarrow \Gamma \mid \{G\}C$$

which requires that all the deposits mentioned in G (either persistent or volatile) occur in Γ. This step corresponds, in R^c, to broadcasting a bitstring which encodes the symbolic term $\{G\}C$. In the bitstring, the deposit names in G are encoded as the identifiers of their corresponding Bitcoin transaction outputs.

Stipulation: committing secrets. After advertisement, committing secrets is done in R^s performing a step such as:

$$\Gamma \mid \{G\}C \rightarrow \Gamma \mid \{G\}C \mid \{\mathsf{A} : a\#N\} \mid \mathsf{A}[\# \triangleright \{G\}C] = \Gamma_1$$

assuming, for simplicity, that G only requires $\mathsf{A}:\mathtt{secret}\, a$. In R^c, this step corresponds to a broadcast of a message $m(C, h, \vec{k})$ which comprises the encoding of the contract advertisement C, the hash h of the secret of A, and the sequence $\vec{k}$ of all the public keys $\mathbf{K}(D, \mathsf{B})$ for any D subterm of C, and any participant B occurring in G. Note that A can obtain $\vec{k}$ by previously exchanging the public keys with other participants in R^c. Message m is then signed using key K_{A}. We further require that, in R^c, the value h was indeed generated by querying the oracle O with some bitstring of length $N + \eta$. If h is not generated through the oracle, or if the required length is shorter than η, the computational commitment would be coherent, instead, with the symbolic commitment $\{\mathsf{A} : a\#\bot\}$, which models a dishonestly chosen secret.

Stipulation: authorizing deposits. The stipulation phase proceeds, in the symbolic model, by providing the authorizations to spend the persistent deposits required by G. This is done in R^s by performing a step such as:

$$\Gamma_1 \rightarrow \Gamma_1 \mid \mathsf{A}[x \triangleright \{G\}C] = \Gamma_2$$

assuming, for simplicity, that G only requires $\mathsf{A}:\,!\, vɃ \,@\, x$. In R^c, this step corresponds to the broadcast of a signature on the T_{init} transaction made with A's key $\hat{K}_{\mathsf{A}}$. This signature authorizes T_{init} to redeem the transaction output corresponding to the symbolic persistent deposit $\langle \mathsf{A}, vɃ\rangle_x$. The unsigned T_{init} transaction is obtained from the compiler, using the hashes and the keys in the commitment messages $m(C, h, \vec{k})$ broadcast in R^c by each participant. Note that an honest participant would sign T_{init} only *after* having exchanged with the other participants the signatures for all the other transactions generated by the compiler.

Stipulation: activating the contract. The stipulation phase is finalised by gathering all the required persistent deposits, and producing an active contract with an initial balance equal to their sum. This is done in R^s by performing a step such as:

$$\Gamma_2 \rightarrow \langle C, vɃ\rangle \mid \Gamma_0 \qquad \text{where } \Gamma = \Gamma_0 \mid \langle \mathsf{A}, vɃ\rangle_x$$

In R^c, this step corresponds to appending T_{init} to the blockchain.

Contract actions. In the symbolic run, once the active contract $\langle C, vɃ\rangle$ is created, it can be executed by performing its actions, causing in R^s steps such as:

$$\langle C, vɃ\rangle \rightarrow \langle C', vɃ\rangle$$

In the computational run, each of these steps corresponds to appending to the blockchain a transaction. The blockchain initially contains a compiler-generated transaction T_C, with an unspent output (say, at index i) corresponding to C. Performing the computational step consists in appending another compiler-generated transaction $\mathsf{T}_{C'}$, which redeems the output i of T_C. In turn, $\mathsf{T}_{C'}$ has an output corresponding to C'. The case of `split` steps, where:

$$\langle C, vɃ\rangle \rightarrow \langle C_1, v_1Ƀ\rangle \mid \cdots \mid \langle C_n, v_nɃ\rangle$$

is similar: $\mathsf{T}_{C'}$ will now have n outputs, corresponding to the n active contracts produced by the symbolic step.

Some symbolic steps can only be performed under certain conditions. For instance, $C = \mathsf{A} : \tau.C'$ (recall that $\tau = \mathtt{reveal}\, \emptyset$) requires A's authorization before it can move forward. In the symbolic model, this authorization is provided by the step:

$$\langle C, vɃ\rangle_x \rightarrow \langle C, vɃ\rangle_x \mid \mathsf{A}[x \triangleright C]$$

In the computational setting, T_C can be redeemed with the signatures of *all* the participants. During the stipulation phase, all such signatures of $\mathsf{T}_{C'}$ are exchanged, *except* for A's one. Participant A can give her authorization at runtime by broadcasting such signature, coherently with the symbolic step above.

Another symbolic step which requires the intervention of a given participant is that of firing a `reveal` prefix. Indeed, in a symbolic run, $C = \mathtt{reveal}\, a.C'$ can only proceed once the secret a has been revealed through a step:

$$\langle C, v\rangle \mid \{\mathsf{A} : a\#N\} \rightarrow \langle C, v\rangle \mid \mathsf{A} : a\#N$$

In the computational run, this corresponds to broadcasting a preimage of the hash value which was broadcast during the stipulation.

After. Symbolic delays trivially correspond to computational ones.

Deposits. Symbolic steps which manage deposits have an immediate computational counterpart. For instance, joining two deposits $\langle \mathsf{A}, v\rangle_x$ and $\langle \mathsf{A}, v'\rangle_y$ into a single deposit $\langle \mathsf{A}, v + v'\rangle_z$ corresponds to appending to the blockchain a transaction with two inputs T_x and T_y and one output, with value $v + v'$. Its witnesses comprise two signatures with $\hat{K}_{\mathsf{A}}$, and its output script verifies a signature with the same key. In the symbolic run, before the join action can be performed, A needs to authorize the spending of x and y, using two distinct steps. In the computational run, these steps correspond to two messages, where A broadcasts the two signatures.

Dually, dividing a deposit $\langle \mathsf{A}, v + v'\rangle$ into two parts $\langle \mathsf{A}, v\rangle$ and $\langle \mathsf{A}, v'\rangle$ corresponds, in the computational run, to appending to the blockchain a transaction with a single input and two outputs. Transforming $\langle \mathsf{A}, v\rangle_x$ into $\langle \mathsf{B}, v\rangle_y$ (i.e., donating a deposit) corresponds,

in the computational run, to appending to the blockchain a transaction having as input T_x, as witness a signature with $\hat{K}_\mathsf{A}$, and a single output of value v, redeemable with $\hat{K}_\mathsf{B}$.

Computational broadcasts. As described above, symbolic authorization or reveal steps correspond to computational broadcasts of suitable messages, such as signatures or hash preimages. Computational participants, however, can also broadcasts other messages, e.g. for exchanging their public keys during stipulation. Further, dishonest computational participants can broadcast any arbitrary bitstring they can compute in PPTIME. Coherence ignores any computational broadcast which does not correspond to any of the above mentioned symbolic steps. That is, such broadcasts correspond to no symbolic steps. Ignoring these messages does not affect the security of contracts, because the other computational messages (in particular, the appended transactions) are enough to reconstruct, from a computational run, the BitML steps in the symbolic run.

Appending non-compiler-generated transactions. A subtle case in the definition of coherence is when *dishonest* participants append transactions to the blockchain. To illustrate the issue, suppose that a dishonest A owns v₿, represented in the symbolic run as a term $\langle \mathsf{A}, v\rangle_x$, and in the computational run as a transaction T_x redeemable with $\hat{K}_\mathsf{A}$. Since A knows her key $\hat{K}_\mathsf{A}$, she can sign an arbitrary transaction T' which redeems v₿ from T_x, and append it to the blockchain. Crucially, T' could be a transaction that can never be generated by the BitML compiler. In such case, it is not possible to precisely match this computational step with a symbolic one. To obtain a correspondence, we let the appending of T' to be coherent with the symbolic *destruction* of the deposit $\langle \mathsf{A}, v\rangle_x$, which makes it disappear from the symbolic configuration. In subsequent steps, coherence will ignore the descendants of T' in the computational run. While, in principle, this loss of information at the symbolic level could allow for computational attacks without a symbolic counterpart, in fact this is not the case, since computational attacks can always be adapted so to have a symbolic counterpart. Indeed, to attack honest participants, A has to stipulate contracts with them: this requires A to put a deposit, computationally represented as a transaction T''. Instead of obtaining T'' from T', which makes T'' unrepresentable at the symbolic level, A can perform symbolically-representable actions to create from T a deposit T''' with the same value of T'', to be used in the computational attack. This adaptation is feasible, because unrepresentable computational actions do not allow the adversary to artificially increase his wealth. Hence, the value of T'' can not exceed the value of T, so making it possible to produce T''' with symbolic actions.

Note that, unlike A's deposits, active contracts involving A can not be destroyed by A in the symbolic run. Hence, if in the computational run A can somehow redeem a transaction T which corresponds to an active contract, using a transaction T' which is not symbolically representable, then such computational step is *not* coherent with any symbolic step. This is intended, since in such case A succeeded in an attack which made the active contract deviate from its symbolic behavior.

Deposits and coherence. The following lemma ensures that deposits in a symbolic run R^s have a correspondent transaction output in any computational run R^c coherent with R^s. A similar correspondence also exists for active contracts (see [20]).

Lemma 1. *Let $R^s \sim_r R^c$. For each deposit $\langle \mathsf{A}, v\rangle$ occurring in Γ_{R^s}, there exists a corresponding unspent transaction output in $\mathbf{B}_{R^c}$ with value v, redeemable with a signature with key $\hat{K}_\mathsf{A}$.*

10 COMPUTATIONAL SOUNDNESS

To state the correctness of the BitML compiler, we need to describe how to convert any symbolic strategy Σ^s_A to a computational strategy $\Sigma^c_\mathsf{A} = \aleph(\Sigma^s_\mathsf{A})$, which realizes the symbolic behaviour in the computational model. Here, we provide the key intuition behind the construction (see [20] for details). Strategy Σ^c_A receives as input a (stripped) computational run R^c. From this, Σ^c_A can reconstruct a (stripped) symbolic run R^s coherent with R^c. At this point, Σ^c_A runs Σ^s_A on R^s, obtaining a set of symbolic actions Λ^s. This is then converted to a set of corresponding computational moves Λ^c, so that performing any of the computational moves will produce an extension of R^c which is still *coherent* with R^s, possibly extended with one of the symbolic moves in Λ^s. This conversion closely follows the definition of coherence.

Our computational soundness result follows (see [20] for its proof). We assume that honest participants have a symbolic strategy, and that their computational strategy is consequently obtained through the mapping $\aleph$ sketched above. Computational soundness establishes that any computational run conforming to the (computational) strategies, with overwhelming probability has a corresponding symbolic run conforming to the (symbolic) strategies.

Theorem 2 (Computational soundness). *Let Σ^s be a set of symbolic strategies for all $\mathsf{A} \in \mathsf{Hon}$. Let Σ^c be a set of computational strategies such that $\Sigma^c_\mathsf{A} = \aleph(\Sigma^s_\mathsf{A})$ for all $\mathsf{A} \in \mathsf{Hon}$, including an arbitrary adversary strategy Σ^c_{Adv}. Fix $k \in \mathbb{N}$. Then, the following set has overwhelming probability:*

$$\left\{ r \;\middle|\; \begin{array}{l} \forall R^c \text{ conforming to } (\Sigma^c, r) \text{ with } |R^c| \le \eta^k : \\ \quad \exists R^s \text{ conforming to } (\Sigma^s, \pi_1(r)) \text{ with } R^s \sim_r R^c \end{array} \right\}$$

In the statement above, r represents the randomness used by all participants, including Adv. Formally, r maps each participant to an infinite string of independent and uniformly distributed bits. Once r is fixed, the behavior of the participants is deterministic, resulting in a single run. Hence, the probability that r drives a run R^c satisfying some property p, as done in the statement, can be seen as the probability that a random run R^c, sampled according to the strategies, satisfies p. The statement considers only computational runs R^c having a *polynomial* length ($\le \eta^k$) with respect to a security parameter η. This is crucial, since the adversary strategy is run at each computational step, hence in the whole run the adversary can run PPTIME algorithms for $|R^c|$ times. Without a polynomial bound on $|R^c|$, the adversary would be able to run algorithms outside PPTIME before the end of the run, breaking the underlying cryptographic primitives. Overall, the theorem states that if we run the computational strategies Σ^c using r, generating a polynomially-bounded run R^c conformant with Σ^c, then, with overwhelming probability, R^c is coherent with some symbolic run R^s. Further, this run can be obtained running the strategies Σ^s using the randomness $\pi_1(r)$. This stands for the sequence of the even-indexed bits in r (this is a technical artifact of our $\aleph$ construction).

Finally, note that Theorem 2 implicitly uses the compiler in two points: the translation $\aleph$ (the obtained computational strategies involve the compilation of the contracts used by the symbolic strategies), and the coherence relation $\sim$.

11 CONCLUSIONS

Our work bridges the gap between the cryptography community, where Bitcoin smart contracts have been investigated first, and the programming languages community. In particular, our computational soundness result guarantees that, if some safety properties are violated at the computational level, then they are also violated at the symbolic level. So, reachability-based symbolic analyses can be soundly used to prove safety properties of smart contracts.

Although BitML can express many of the smart contracts appeared in literature [14], it has some limitations. For instance, it cannot express contingent payments, where a participant A promises to pay B for a value x satisfying a predicate chosen by A (e.g., x is a prime factor of a given large number). Contingent payments can be implemented in Bitcoin similarly to timed commitments: A pays a deposit, which is taken by B after revealing a preimage of $H(x)$ which satisfies the predicate. An off-chain protocol [17] (which exploits zero-knowledge proofs) is used to guarantee that $H(x)$ is indeed the hash of a value x satisfying the predicate (note that, in the Bitcoin scripting language, one can only check trivial predicates, like e.g. equality). BitML could be extended to express contingent payments, by exploiting zero-knowledge proofs similar to those in [17] in the stipulation phase. This would allow our compiler to only generate standard Bitcoin transactions. Another kind of contracts which are not expressible in BitML are those for which one cannot pre-determine at compile time, a *finite* set of transactions, or of signatures, or of execution steps. This is the case, e.g., of crowdfunding contracts [14], where participants invest some money until a given threshold is reached. Here, we do not statically know neither the number of participants, nor their identities, so it is not possible to statically produce (and pre-sign) a set of transactions, as required by BitML. Extending BitML to express this kind of contracts — while preserving our compilation technique — would require suitable extensions of Bitcoin transactions. For instance, recursion could be obtained via extensions similar to those proposed in [49, 53].

Acknowledgments. This work is partially supported by Aut. Reg. of Sardinia projects "Sardcoin" and "Smart collaborative engineering".

REFERENCES

[1] 2012. Bitcoin wiki - Contracts. https://en.bitcoin.it/wiki/Contract.
[2] 2015. Bitcoin developer guide - Escrow and arbitration. https://bitcoin.org/en/developer-guide#escrow-and-arbitration.
[3] 2016. Understanding the DAO attack. http://www.coindesk.com/understanding-dao-hack-journalists/.
[4] 2017. IVY. https://docs.ivy-lang.org/bitcoin/.
[5] 2017. Parity Wallet Security Alert. https://paritytech.io/blog/security-alert.html.
[6] 2017. A Postmortem on the Parity Multi-Sig Library Self-Destruct. https://goo.gl/Kw3gXi.
[7] 2018. BALZaC: Bitcoin Abstract Language, analyZer and Compiler. https://blockchain.unica.it/balzac/.
[8] Martín Abadi and Phillip Rogaway. 2007. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption). *J. Cryptology* 20, 3 (2007), 395. https://doi.org/10.1007/s00145-007-0203-0
[9] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *EuroSys*. 30:1–30:15. https://doi.org/10.1145/3190508.3190538
[10] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2014. Fair Two-Party Computations via Bitcoin Deposits. In *Financial Cryptography Workshops (LNCS)*, Vol. 8438. Springer, 105–121. https://doi.org/10.1007/978-3-662-44774-1_8
[11] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2014. Secure Multiparty Computations on Bitcoin. In *IEEE S & P*. 443–458. https://doi.org/10.1109/SP.2014.35 First appeared on Cryptology ePrint Archive, http://eprint.iacr.org/2013/784.
[12] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. 2017. Hijacking Bitcoin: Routing Attacks on Cryptocurrencies. In *IEEE Symp. on Security and Privacy*. 375–392. https://doi.org/10.1109/SP.2017.29
[13] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust (POST) (LNCS)*, Vol. 10204. Springer, 164–186. https://doi.org/10.1007/978-3-662-54455-6_8
[14] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, and Roberto Zunino. 2018. SoK: unraveling Bitcoin smart contracts. In *Principles of Security and Trust (POST) (LNCS)*, Vol. 10804. Springer, 217–242. https://doi.org/10.1007/978-3-319-89722-6
[15] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. 2018. A formal model of Bitcoin transactions. In *Financial Cryptography and Data Security*.
[16] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. 2017. Bitcoin as a Transaction Ledger: A Composable Treatment. In *CRYPTO*. 324–356. https://doi.org/10.1007/978-3-319-63688-7_11
[17] Waclaw Banasik, Stefan Dziembowski, and Daniel Malinowski. 2016. Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts. In *ESORICS (LNCS)*, Vol. 9879. Springer, 261–280. https://doi.org/10.1007/978-3-319-45741-3_14
[18] Massimo Bartoletti and Livio Pompianu. 2017. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *Financial Cryptography Workshops (LNCS)*, Vol. 10323. Springer, 494–509. https://doi.org/10.1007/978-3-319-70278-0_31
[19] Massimo Bartoletti and Roberto Zunino. 2017. Constant-deposit multiparty lotteries on Bitcoin. In *Financial Cryptography Workshops (LNCS)*, Vol. 10323. Springer. https://doi.org/10.1007/978-3-319-70278-0
[20] Massimo Bartoletti and Roberto Zunino. 2018. BitML: a Calculus for Bitcoin Smart Contracts. *IACR Cryptology ePrint Archive* 2018 (2018), 122. http://eprint.iacr.org/2018/122
[21] Mihir Bellare and Phillip Rogaway. 1993. Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols. In *ACM Conference on Computer and Communications Security*. ACM, 62–73. https://doi.org/10.1145/168588.168596
[22] Iddo Bentov and Ranjit Kumaresan. 2014. How to Use Bitcoin to Design Fair Protocols. In *CRYPTO (LNCS)*, Vol. 8617. Springer, 421–439. https://doi.org/10.1007/978-3-662-44381-1_24
[23] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cedric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Beguelin. 2016. Formal Verification of Smart Contracts. In *PLAS*.
[24] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. 2017. Findel: Secure Derivative Contracts for Ethereum. In *Financial Cryptography Workshops (LNCS)*, Vol. 10323. Springer, 453–467. https://doi.org/10.1007/978-3-319-70278-0_28
[25] BitFury group. 2015. Smart Contracts on Bitcoin Blockchain. http://bitfury.com/content/5-white-papers-research/contracts-1.1.1.pdf.
[26] Dan Boneh and Moni Naor. 2000. Timed Commitments. In *CRYPTO (LNCS)*, Vol. 1880. Springer, 236–254. https://doi.org/10.1007/3-540-44598-6
[27] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. 2015. SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *IEEE S & P*. 104–121. https://doi.org/10.1109/SP.2015.14
[28] Vitalik Buterin. 2013. Ethereum: a next generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper.
[29] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Yaron Velner. 2018. Quantitative Analysis of Smart Contracts. In *ESOP*. 739–767. https://doi.org/10.1007/978-3-319-89884-1_26
[30] Karl Crary and Michael J. Sullivan. 2015. Peer-to-peer affine commitment using Bitcoin. In *ACM Conf. on Programming Language Design and Implementation*. 479–488. https://doi.org/10.1145/2737924.2737997
[31] Christian Decker and Roger Wattenhofer. 2015. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In *Stabilization, Safety, and Security of Distributed Systems (SSS) (LNCS)*, Vol. 9212. Springer, 3–18. https://doi.org/10.1007/978-3-319-21741-3_1
[32] Sergi Delgado-Segura, Cristina Pérez-Solà, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomartí. 2017. A fair protocol for data trading based on Bitcoin

transactions. *Future Generation Computer Systems* (2017). https://doi.org/10.1016/j.future.2017.08.021
[33] Ittay Eyal and Emin Gün Sirer. 2014. Majority Is Not Enough: Bitcoin Mining Is Vulnerable. In *Financial Cryptography (LNCS)*, Vol. 8437. Springer, 436–454. https://doi.org/10.1007/978-3-662-45472-5_28
[34] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The Bitcoin Backbone Protocol: Analysis and Applications. In *EUROCRYPT (LNCS)*, Vol. 9057. Springer, 281–310. https://doi.org/10.1007/978-3-662-46803-6_10
[35] David M. Goldschlag, Stuart G. Stubblebine, and Paul F. Syverson. 2010. Temporarily hidden bit commitment and lottery applications. *Int. J. Inf. Sec.* 9, 1 (2010), 33–50. https://doi.org/10.1007/s10207-009-0094-1
[36] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. Foundations and Tools for the Static Analysis of Ethereum Smart Contracts. In *CAV (LNCS)*, Vol. 10981. Springer, 51–78. https://doi.org/10.1007/978-3-319-96145-3_4
[37] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust (POST) (LNCS)*, Vol. 10804. Springer, 243–269. https://doi.org/10.1007/978-3-319-89722-6_10
[38] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography Workshops (LNCS)*, Vol. 10323. Springer, 520–535. https://doi.org/10.1007/978-3-319-70278-0_33
[39] Simon L. Peyton Jones, Jean-Marc Eber, and Julian Seward. 2000. Composing contracts: an adventure in financial engineering, functional pearl. In *International Conference on Functional Programming (ICFP)*. 280–292. https://doi.org/10.1145/351240.351267
[40] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *IEEE Symp. on Security and Privacy*. 839–858. https://doi.org/10.1109/SP.2016.55
[41] Ranjit Kumaresan and Iddo Bentov. 2014. How to Use Bitcoin to Incentivize Correct Computations. In *ACM CCS*. 30–41. https://doi.org/10.1145/2660267.2660380
[42] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. 2015. How to Use Bitcoin to Play Decentralized Poker. In *ACM CCS*. 195–206. https://doi.org/10.1145/2810103.2813712
[43] Eric Lombrozo, Johnson Lau, and Pieter Wuille. 2015. Segregated Witness (Consensus layer). BIP 141, https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki.
[44] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *ACM CCS*. 254–269. https://doi.org/10.1145/2976749.2978309
[45] Anastasia Mavridou and Aron Laszka. 2018. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In *Financial Cryptography and Data Security*.
[46] Gregory Maxwell. 2016. The first successful Zero-Knowledge Contingent Payment. https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/.
[47] Andrew Miller and Iddo Bentov. 2017. Zero-Collateral Lotteries in Bitcoin and Ethereum. In *EuroS&P Workshops*. 4–13. https://doi.org/10.1109/EuroSPW.2017.44
[48] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. 2017. Sprites: Payment Channels that Go Faster than Lightning. *CoRR* abs/1702.05812 (2017). arXiv:1702.05812 http://arxiv.org/abs/1702.05812
[49] Malte Möser, Ittay Eyal, and Emin Gün Sirer. 2016. Bitcoin covenants. In *Financial Cryptography Workshops (LNCS)*, Vol. 9604. Springer, 126–141. https://doi.org/10.1007/978-3-662-53357-4_9
[50] Xavier Nicollin and Joseph Sifakis. 1991. An Overview and Synthesis on Timed Process Algebras. In *CAV*. 376–398. https://doi.org/10.1007/3-540-55179-4_36
[51] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.
[52] Russell O'Connor. 2017. Simplicity: A New Language for Blockchains. In *PLAS*. http://arxiv.org/abs/1711.03028
[53] Russell O'Connor and Marta Piekarska. 2017. Enhancing Bitcoin transactions with covenants. In *Financial Cryptography Workshops (LNCS)*, Vol. 10323. Springer. https://doi.org/10.1007/978-3-319-70278-0_12
[54] Joseph Poon and Thaddeus Dryja. 2015. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://lightning.network/lightning-network-paper.pdf
[55] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR* abs/1801.00687 (2018).
[56] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F*. In *POPL*. https://doi.org/10.1145/2837614.2837655
[57] Paul F. Syverson. 1998. Weakly Secret Bit Commitment: Applications to Lotteries and Fair Exchange. In *IEEE CSFW*. 2–13. https://doi.org/10.1109/CSFW.1998.683149
[58] Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday* 2, 9 (1997). http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/548.
[59] Petar Tsankov, Andrei Marian Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. *CoRR* abs/1806.01143 (2018).

A APPENDIX

The semantics of BitML is organised in two layers: a bottom layer, taking the form of an LTS between (untimed) *configurations*, and a top layer, in the form of a timed LTS between *timed configurations*. We assume that $(|, 0)$ is a commutative monoid. Indexed parallel compositions are denoted with $\|$. Authorizations are as follows:

$\chi ::=$	authorization to …
$\# \triangleright \{G\}C$	commit secrets to stipulate $\{G\}C$
$\mid x \triangleright \{G\}C$	spend x to stipulate $\{G\}C$
$\mid x \triangleright D$	take branch D
$\mid x, y \triangleright \langle \mathsf{A}, v \rangle$	join deposit x with y into a deposit for A
$\mid x \triangleright \langle \mathsf{A}, v \rangle, \langle \mathsf{A}, v' \rangle$	divide a deposit x in two deposits for A
$\mid x \triangleright \mathsf{B}$	donate deposit x to B
$\mid \vec{x}, i \triangleright y$	destroy i-th deposits in $\vec{x}$ through y

The LTS of untimed configurations is given in Figure 2. The function cv from labels α to sets of names is defined as: $cv(split(y)) = cv(put(\vec{x}, \vec{a}, y)) = cv(withdraw(\mathsf{A}, v, y)) = \{y\}$, otherwise $cv(\alpha)$ is empty. We further assume that a transition $\Gamma \to \Gamma \mid \mathsf{A}[\chi]$ is possible only if $\mathsf{A}[\chi]$ is not already present in Γ.

Rule [Dep-AuthJoin] allows A to authorize the merge of two deposits x, y into a single one, creating the needed authorization. The label $\mathsf{A} : \cdots$ records that only A can perform this move. [Dep-Join] uses this authorization to create a single deposit z of A. [Dep-AuthDivide] and [Dep-Divide] act similarly, allowing a deposit of A to be divided in two parts. [Dep-AuthDonate] and [Dep-Donate] allow A to transfer one of her deposits to another participant. [Dep-AuthDestroy] and [Dep-Destroy] allow a set of participants to destroy a set of deposits $x_1 \cdots x_n$. To do that, first each participant A_i must provide the needed authorization $\mathsf{A}_i[\vec{x}, i \triangleright y]$ for their own deposit x_i. When all the authorizations have been collected, [Dep-destroy] eliminates the deposits. The last pair of rules are needed to represent the fact that computational participants can append transactions with no symbolic counterpart. To achieve a meaningful symbolic-computational correspondence, appending such transactions is rendered by [Dep-destroy]. Note that [C-Advertise] requires that at least one of the participants involved in each stipulation is honest (one of the weakest assumptions in cryptographic protocols). The same effect of running contracts among dishonest participants can still be obtained by redistributing funds through the rules for deposits. Hence, this side condition does not affect the power of the adversary. A further motivation for the side condition is that the correctness of our compiler will rely on this assumption. Rule [C-AuthCommit] allows dishonest participants to choose an "invalid" length $\perp$ for their committed secrets. This reflects the fact that, in the computational model, A commits to a secret by broadcasting a bitstring, meant to be the hash of the secret. If A is dishonest, she could broadcast an arbitrary bitstring w, preventing herself later on to reveal a preimage of w. Similarly, the length $\perp$ prevents the reveal action in the symbolic model.

$$\frac{}{\langle \mathsf{A}, v\rangle_x \mid \langle \mathsf{A}, v'\rangle_y \mid \Gamma \xrightarrow{\mathsf{A}:x,y} \langle \mathsf{A}, v\rangle_x \mid \langle \mathsf{A}, v'\rangle_y \mid \mathsf{A}[x, y \rhd \langle \mathsf{A}, v + v'\rangle] \mid \Gamma}\ [\textsc{Dep-AuthJoin}]$$

$$\frac{\Gamma = \mathsf{A}[x, y \rhd \langle \mathsf{A}, v + v'\rangle] \mid \mathsf{A}[y, x \rhd \langle \mathsf{A}, v + v'\rangle] \mid \Gamma' \quad z \text{ fresh}}{\langle \mathsf{A}, v\rangle_x \mid \langle \mathsf{A}, v'\rangle_y \mid \Gamma \xrightarrow{join(x,y)} \langle \mathsf{A}, v + v'\rangle_z \mid \Gamma'}\ [\textsc{Dep-Join}]$$

$$\frac{}{\langle \mathsf{A}, v + v'\rangle_x \mid \Gamma \xrightarrow{\mathsf{A}:x,v,v'} \langle \mathsf{A}, v + v'\rangle_x \mid \mathsf{A}[x \rhd \langle \mathsf{A}, v\rangle, \langle \mathsf{A}, v'\rangle] \mid \Gamma}\ [\textsc{Dep-AuthDivide}]$$

$$\frac{\Gamma = \mathsf{A}[x \rhd \langle \mathsf{A}, v\rangle, \langle \mathsf{A}, v'\rangle] \mid \Gamma' \quad y, y' \text{ fresh}}{\langle \mathsf{A}, v + v'\rangle_x \mid \Gamma \xrightarrow{divide(x,v,v')} \langle \mathsf{A}, v\rangle_y \mid \langle \mathsf{A}, v'\rangle_{y'} \mid \Gamma'}\ [\textsc{Dep-Divide}]$$

$$\frac{}{\langle \mathsf{A}, v\rangle_x \mid \Gamma \xrightarrow{\mathsf{A}:x,\mathsf{B}} \langle \mathsf{A}, v\rangle_x \mid \mathsf{A}[x \rhd \mathsf{B}] \mid \Gamma}\ [\textsc{Dep-AuthDonate}] \qquad \frac{\Gamma = \mathsf{A}[x \rhd \mathsf{B}] \mid \Gamma' \quad y \text{ fresh}}{\langle \mathsf{A}, v\rangle_x \mid \Gamma \xrightarrow{donate(x,\mathsf{B})} \langle \mathsf{B}, v\rangle_y \mid \Gamma'}\ [\textsc{Dep-Donate}]$$

$$\frac{\vec{x} = x_1 \cdots x_n \quad j \in 1..n \quad y \text{ fresh (except in destroy authorizations for } \vec{x})}{\left(\|_{i\in 1..n}\langle \mathsf{A}_i, v_i\rangle_{x_i}\right) \mid \Gamma \xrightarrow{\mathsf{A}_j:\vec{x},j} \left(\|_{i\in 1..n}\langle \mathsf{A}_i, v_i\rangle_{x_i}\right) \mid \mathsf{A}_j[\vec{x}, j \rhd y] \mid \Gamma}\ [\textsc{Dep-AuthDestroy}]$$

$$\frac{\vec{x} = x_1 \cdots x_n \quad \Gamma = \left(\|_{i\in 1..n}\, \mathsf{A}_i[\vec{x}, i \rhd y]\right) \mid \Gamma'}{\left(\|_{i\in 1..n}\langle \mathsf{A}_i, v_i\rangle_{x_i}\right) \mid \Gamma \xrightarrow{destroy(\vec{x})} \Gamma'}\ [\textsc{Dep-Destroy}]$$

$$\frac{\begin{array}{ll}\{G\}C \text{ contains at least one participant in Hon} & \Gamma \text{ contains } \langle \mathsf{A}_i, v_i\rangle_{x_i} \text{ for all } \mathsf{A}_i : !\, v_i \,@\, x_i \text{ in } \{G\}C \\ a \text{ fresh, for each } \mathsf{A} : \mathtt{secret}\ a \text{ in } G & \Gamma \text{ contains } \langle \mathsf{A}_i, v_i\rangle_{x_i} \text{ for all } \mathsf{A}_i : ?\, v_i \,@\, x_i \text{ in } \{G\}C\end{array}}{\Gamma \xrightarrow{advertise(\{G\}C)} \{G\}C \mid \Gamma}\ [\textsc{C-Advertise}]$$

$$\frac{\begin{array}{ll}a_1 \cdots a_k \text{ secrets of A in } G & \\ \forall i \in 1..k : \nexists N : \{\mathsf{A} : a_i \# N\} \in \Gamma & \forall i \in 1..k : N_i \in \begin{cases}\mathbb{N} & \text{if } \mathsf{A} \in \mathsf{Hon} \\ \mathbb{N} \cup \{\bot\} & \text{otherwise}\end{cases} \\ \Delta = \{\mathsf{A} : a_1 \# N_1\} \mid \cdots \mid \{\mathsf{A} : a_k \# N_k\} & \end{array}}{\{G\}C \mid \Gamma \xrightarrow{\mathsf{A}:\{G\}C,\Delta} \{G\}C \mid \Gamma \mid \Delta \mid \mathsf{A}[\# \rhd \{G\}C]}\ [\textsc{C-AuthCommit}]$$

$$\frac{\Gamma \text{ contains } \mathsf{B}[\# \rhd \{G\}C] \text{ for all B in } G \qquad G = \mathsf{A} : !\, v \,@\, x \mid \cdots}{\{G\}C \mid \Gamma \xrightarrow{\mathsf{A}:\{G\}C,x} \{G\}C \mid \Gamma \mid \mathsf{A}[x \rhd \{G\}C]}\ [\textsc{C-AuthInit}]$$

$$\frac{G = \left(\|_{i\in I}\mathsf{A}_i : !\, v_i \,@\, x_i\right) \mid \left(\|_{i\in J}\mathsf{B}_i : ?\, v'_i \,@\, y_i\right) \mid \left(\|_{i\in K}\mathsf{C}_i : \mathtt{secret}\ a_i\right) \quad x \text{ fresh}}{\{G\}C \mid \Gamma \mid \left(\|_{i\in I}\langle \mathsf{A}_i, v_i\rangle_{x_i}\right) \mid \left(\|_{A\in G}\mathsf{A}[\# \rhd \{G\}C]\right) \mid \left(\|_{i\in I}\mathsf{A}_i[x_i \rhd \{G\}C]\right) \xrightarrow{init(G,C)} \langle C, \textstyle\sum_{i\in I} v_i\rangle_x \mid \Gamma}\ [\textsc{C-Init}]$$

$$\frac{\vec{v} = v_1 \cdots v_k \qquad \vec{C} = C_1 \cdots C_k \qquad \sum_{i=1}^{k} v_k = v' \qquad x_1 \cdots x_k \text{ fresh}}{\langle \mathtt{split}\ \vec{v} \to \vec{C}, v'\rangle_y \mid \Gamma \xrightarrow{split(y)} \left(\|_{i=1}^{k}\langle C_i, v_i\rangle_{x_i}\right) \mid \Gamma}\ [\textsc{C-Split}]$$

$$\frac{N \neq \bot}{\{\mathsf{A} : a\#N\} \xrightarrow{\mathsf{A}:a} \mathsf{A} : a\#N}\ [\textsc{C-AuthRev}] \qquad \frac{\begin{array}{lll}\vec{x} = x_1 \cdots x_m & \Gamma = \|_{i=1}^{m}\langle \mathsf{A}_i, v_i\rangle_{x_i} & z \text{ fresh} \\ \vec{a} = a_1 \cdots a_n & \Delta = \|_{i=1}^{n}\mathsf{B}_i : a_i\#N_i & [\![p]\!]_\Delta = true\end{array}}{\langle \mathtt{put}\ \vec{x}\ \&\ \mathtt{reveal}\ \vec{a}\ \mathtt{if}\ p.\ C, v\rangle_y \mid \Gamma \mid \Delta \mid \Gamma' \xrightarrow{put(\vec{x},\vec{a},y)} \langle C, v + \textstyle\sum_{i=1}^{m} v_i\rangle_z \mid \Delta \mid \Gamma'}\ [\textsc{C-PutRev}]$$

$$\frac{x \text{ fresh}}{\langle \mathtt{withdraw}\ \mathsf{A}, v\rangle_y \mid \Gamma \xrightarrow{withdraw(\mathsf{A},v,y)} \langle \mathsf{A}, v\rangle_x \mid \Gamma}\ [\textsc{C-Withdraw}] \qquad \frac{D \equiv \mathsf{A} : D'}{\langle D + C, v\rangle_x \mid \Gamma \xrightarrow{\mathsf{A}:x,D} \langle D + C, v\rangle_x \mid \mathsf{A}[x \rhd D] \mid \Gamma}\ [\textsc{C-AuthControl}]$$

$$\frac{D \equiv \mathsf{A}_1 : \cdots : \mathsf{A}_k : \mathtt{after}\ t_1 : \cdots : \mathtt{after}\ t_m : D' \quad D' \not\equiv \mathsf{A} : \cdots \quad D' \not\equiv \mathtt{after}\ t : \cdots \quad \langle D', v\rangle_x \mid \Gamma \xrightarrow{\alpha} \Gamma' \quad x \in cv(\alpha)}{\langle D + C, v\rangle_x \mid \|_{i=1}^{k}\mathsf{A}_i[x \rhd D] \mid \Gamma \xrightarrow{\alpha} \Gamma'}\ [\textsc{C-Control}]$$

$$[\![true]\!]_\Gamma = true \qquad [\![p_1 \wedge p_2]\!]_\Gamma = [\![p_1]\!]_\Gamma \text{ and } [\![p_2]\!]_\Gamma \qquad [\![\neg p]\!]_\Gamma = \text{not } [\![p]\!]_\Gamma \qquad [\![E_1 \circ E_2]\!]_\Gamma = [\![E_1]\!]_\Gamma \circ [\![E_2]\!]_\Gamma \quad (\circ \in \{=, <\})$$

$$[\![N]\!]_\Gamma = N \qquad [\![|a|]\!]_\Gamma = N \text{ if } \Gamma \text{ contains } \mathsf{A} : a\#N \qquad [\![E_1 \bullet E_2]\!]_\Gamma = [\![E_1]\!]_\Gamma \bullet [\![E_2]\!]_\Gamma \quad (\bullet \in \{+, -\})$$

Figure 2: Semantics of untimed configurations.

$$\frac{\Gamma \xrightarrow{\alpha} \Gamma' \quad cv(\alpha) = \emptyset}{\Gamma \mid t \xrightarrow{\alpha} \Gamma' \mid t}\ [\textsc{Action}] \qquad \frac{\delta > 0}{\Gamma \mid t \xrightarrow{\delta} \Gamma \mid t + \delta}\ [\textsc{Delay}]$$

$$\frac{D \equiv \texttt{after}\, t_1 : \cdots : \texttt{after}\, t_m : D' \quad D' \not\equiv \texttt{after}\, t' : \cdots \quad \langle D, v\rangle_x \mid \Gamma \xrightarrow{\alpha} \Gamma' \quad x \in cv(\alpha) \quad t \geq t_1, \ldots, t_m}{\langle D + C, v\rangle_x \mid \Gamma \mid t \xrightarrow{\alpha} \Gamma' \mid t}\ [\textsc{Timeout}]$$

Figure 3: Semantics of timed configurations.

$$\frac{\begin{array}{l} G = \left(\|_{i\in I} A_i : ?\, v_i \,@\, x_i\right) \mid \left(\|_{i\in J} B_i : !\, v'_i \,@\, y_i\right) \mid \left(\|_{i\in K} C_i : \texttt{secret}\, a_i\right) \\ C = \sum_{i=1}^{m} D_i \qquad v = \sum_{i\in J} v'_i \\ e_i = \mathbb{B}_{\text{out}}(D_i)\ \ (\forall i \in 1..m) \qquad \vec{x} = \uplus_{i=1}^{m} \text{fv}(e_i) \\ \mathcal{T}_i = \mathbb{B}_{\text{D}}(D_i, D_i, \mathsf{T}_{init}, 0, v, \text{PartG}, 0)\ \ (\forall i \in 1..m) \end{array}}{\mathbb{B}_{\text{adv}}(\{G\}C) = \mathsf{T}_{init}\, \mathcal{T}_1 \cdots \mathcal{T}_m}$$

T_{init}
in: $i \mapsto \text{txout}(y_i)\ \ (\forall i \in J)$
wit: $\bot$
out: $\left(\lambda \vec{x}.\ \bigvee_{i=1}^{m} e_i,\ v\right)$

$$\frac{\begin{array}{l} C = \sum_{i=1}^{m} D_i \qquad I = \{z_1, \ldots, z_k\} \\ e_i = \mathbb{B}_{\text{out}}(D_i)\ \ (\forall i \in 1..m) \\ \vec{x} = \uplus_{i=1}^{m} \text{fv}(e_i) \\ \mathcal{T}_i = \mathbb{B}_{\text{D}}(D_i, D_i, \mathsf{T}_C, 0, v, \text{PartG}, 0)\ \ (\forall i \in 1..m) \end{array}}{\mathbb{B}_{\text{C}}(C, D_p, \mathsf{T}, o, v, I, \mathcal{P}, t) = \mathsf{T}_C\, \mathcal{T}_1 \cdots \mathcal{T}_m}$$

T_C
in: $0 \mapsto (\mathsf{T}, o),\ \ i \mapsto \text{txout}(z_i)\ \ (\forall i \in 1..k)$
wit: $0 \mapsto sig_{\text{K}(D_p,\mathcal{P})},\ \ i \mapsto sig_{\text{K}(\text{part}(z_i))}\ \ (\forall i \in 1..k)$
out: $\left(\lambda \vec{x}.\ \bigvee_{i=1}^{m} e_i,\ v\right)$
absLock: t

$$\frac{D \not\equiv A_1 : \cdots : A_n : \texttt{after}\, t_1 : \cdots : \texttt{after}\, t_k : \texttt{put}\, \vec{z}\, \&\, \texttt{reveal}\, \vec{a}\, \texttt{if}\, p.\ C \quad \vec{\varsigma}\ \text{fresh}}{\mathbb{B}_{\text{out}}(D) = \text{versig}_{\text{K}(D,\text{PartG})}(\vec{\varsigma})}$$

$$\frac{D \equiv A_1 : \cdots : A_n : \texttt{after}\, t_1 : \cdots : \texttt{after}\, t_k : \texttt{put}\, \vec{z}\, \&\, \texttt{reveal}\, \vec{a}\, \texttt{if}\, p.\ C \quad \vec{a} = a_1 \cdots a_m \quad \vec{\varsigma}, b_1 \cdots b_m\ \text{fresh}}{\mathbb{B}_{\text{out}}(D) = \text{versig}_{\text{K}(D,\text{PartG})}(\vec{\varsigma}) \ \wedge\ \mathbb{B}(p) \ \wedge\ \bigwedge_{i=1}^{m} \text{H}(b_i) = \text{sechash}(a_i) \ \wedge\ |b_i| \geq \eta}$$

$$\frac{D = \texttt{withdraw}\, A}{\mathbb{B}_{\text{D}}(D, D_p, \mathsf{T}, o, v, \mathcal{P}, t) = \{\text{in} : (\mathsf{T}, o),\ \ \text{wit} : sig_{\text{K}(D_p,\mathcal{P})},\ \ \text{out} : \left(\lambda\varsigma.\ \text{versig}_{\text{K}(A)}(\varsigma),\ v\right),\ \ \text{absLock} : t\}}$$

$$\frac{D = \texttt{put}\, \vec{z}\, \&\, \texttt{reveal}\, \vec{a}\, \texttt{if}\, p.\ C \quad v' = v + \sum_{x\in\vec{z}} \text{val}(x)}{\mathbb{B}_{\text{D}}(D, D_p, \mathsf{T}, o, v, \mathcal{P}, t) = \mathbb{B}_{\text{C}}(C, D_p, \mathsf{T}, o, v', \vec{z}, \mathcal{P}, t)} \qquad \frac{D = \texttt{split}\, \vec{v} \to \vec{C} \quad \vec{v} = v_1 \cdots v_k \quad \sum_{i=1}^{k} v_i \leq v}{\mathbb{B}_{\text{D}}(D, D_p, \mathsf{T}, o, v, \mathcal{P}, t) = \mathbb{B}_{\text{par}}(\vec{C}, D_p, \mathsf{T}, o, \vec{v}, \mathcal{P}, t)}$$

$$\frac{D = A : D'}{\mathbb{B}_{\text{D}}(D, D_p, \mathsf{T}, o, v, \mathcal{P}, t) = \mathbb{B}_{\text{D}}(D', D_p, \mathsf{T}, o, v, \mathcal{P} \setminus \{A\}, t)} \qquad \frac{D = \texttt{after}\, t' : D'}{\mathbb{B}_{\text{D}}(D, D_p, \mathsf{T}, o, v, \mathcal{P}, t) = \mathbb{B}_{\text{D}}(D', D_p, \mathsf{T}, o, v, \mathcal{P}, \max\{t, t'\})}$$

$$\frac{\begin{array}{l} \vec{C} = C_1 \cdots C_n \qquad C_i = \sum_{j=1}^{k_i} D_{i,j}\ \ (\forall i \in 1..n) \\ \vec{v} = v_1 \cdots v_n \qquad e_{i,j} = \mathbb{B}_{\text{out}}(D_{i,j})\ \ (\forall i \in 1..n, j \in 1..k_i) \\ \vec{x}_i = \uplus_{j=1}^{k_i} \text{fv}(e_{i,j})\ \ (\forall i \in 1..n) \\ \mathcal{T}_{i,j} = \mathbb{B}_{\text{D}}(D_{i,j}, D_{i,j}, \mathsf{T}_C, i-1, v_i, \text{PartG}, 0)\ \ (\forall i \in 1..n, j \in 1..k_i) \end{array}}{\mathbb{B}_{\text{par}}(\vec{C}, D_p, \mathsf{T}, o, \vec{v}, \mathcal{P}, t) = \mathsf{T}_C(\mathcal{T}_{i,j})_{i\in 1..n,\, j\in 1..k_i}}$$

T_C
in: (T, o)
wit: $sig_{\text{K}(D_p,\mathcal{P})}$
out: $i - 1 \mapsto \left(\lambda \vec{x}_i.\ \bigvee_{j=1}^{k_i} e_{i,j},\ v_i\right)\ \ (\forall i \in 1..n)$
absLock: t

$$\mathbb{B}(true) = true \qquad \mathbb{B}(p_1 \wedge p_2) = \mathbb{B}(p_1) \wedge \mathbb{B}(p_2) \qquad \mathbb{B}(\neg p) = \neg\mathbb{B}(p) \qquad \mathbb{B}(e_1 \circ e_2) = \mathbb{B}(e_1) \circ \mathbb{B}(e_2)$$

$$\mathbb{B}(N) = N \qquad \mathbb{B}(|a|) = |a| - \eta \qquad \mathbb{B}(e_1 \bullet e_2) = \mathbb{B}(e_1) \bullet \mathbb{B}(e_2)$$

Figure 4: Formalization of the BitML compiler. The compiler relies on the following parameters, which depend on G and C: *(i)* PartG is the set of all participants occurring in G; *(ii)* part maps deposit names in G to the corresponding participants (e.g., $\text{part}(x) = A$ if $A : ?\, v \,@\, x$ in G); *(iii)* txout maps deposit names in G to the corresponding Bitcoin transaction output (T, o); *(iv)* val maps deposit names in G to the value contained in the deposit (e.g., $\text{val}(x) = v$ if $A : ?\, v \,@\, x$ in G); *(v)* sechash maps secret names in G to the corresponding committed hashes.

Timed commitment We show two computations of the timed commitment contract TC introduced in Section 2, using

$$G \;=\; \mathsf{A} : !\; 1\ddot{\mathbb{B}} \,@\, x \mid \mathsf{A} : \mathtt{secret}\; a \mid \mathsf{B} : !\; 0\ddot{\mathbb{B}} \,@\, y$$

as precondition. Let:

$$\Gamma \;=\; \langle \mathsf{A}, 1\ddot{\mathbb{B}} \rangle_x \mid \langle \mathsf{B}, 0\ddot{\mathbb{B}} \rangle_y$$

Assuming that A is honest, a possible computation where A reveals her secret and then redeems the deposit is the following (here time is immaterial):

$$\begin{aligned}
&\Gamma \to \Gamma \mid \{G\}TC = \Gamma_1 && (1)\\
&\quad \to \Gamma_1 \mid \{\mathsf{A} : a\#N\} \mid \mathsf{A}[\# \rhd \{G\}TC] = \Gamma_2 && (2)\\
&\quad \to \Gamma_2 \mid \mathsf{B}[\# \rhd \{G\}TC] = \Gamma_3 && (3)\\
&\quad \to \Gamma_3 \mid \mathsf{A}[x \rhd \{G\}TC] = \Gamma_4 && (4)\\
&\quad \to \Gamma_4 \mid \mathsf{B}[y \rhd \{G\}TC] && (5)\\
&\quad \to \langle TC, 1\ddot{\mathbb{B}} \rangle_{x_1} \mid \{\mathsf{A} : a\#N\} \;=\; \Gamma' && (6)\\
&\quad \to \langle TC, 1\ddot{\mathbb{B}} \rangle_{x_1} \mid \mathsf{A} : a\#N && (7)\\
&\quad \to \langle \mathtt{withdraw}\ \mathsf{A}, 1\ddot{\mathbb{B}} \rangle_{x_2} \mid \mathsf{A} : a\#N && (8)\\
&\quad \to \langle \mathsf{A}, 1\ddot{\mathbb{B}} \rangle_{x_3} \mid \mathsf{A} : a\#N && (9)
\end{aligned}$$

Step (1) advertises $\{G\}TC$. This is possible because both deposits x and y (required by G) are available in Γ. At step (2), A commits to a secret: its length N is a natural, since A is honest. At step (3) also B does his commitment (empty, since G does not require any secrets from B). At steps (4)-(5), A and B give their authorization to stipulate TC, by providing their authorizations to spend the deposits x and y, respectively. At step (6) the contract TC becomes stipulated. After this step, the bitcoins in the deposits x and y are transferred to the contract. At step (7), A reveals her secret (and consequently, also its length N). After that, the action `reveal` a is performed at step (8), reducing the contract to `withdraw` A, and discarding the `after` branch. Finally, step (9) performs the `withdraw` A action, producing a fresh deposit x_3 with $1\ddot{\mathbb{B}}$ redeemable by A.

We also show a computation where A does not reveal her secret, and B waits until $t' > t$ to redeem A's deposit. Starting from the configuration Γ' at time 0, we have the following steps:

$$\Gamma' \mid 0 \;\to\; \Gamma' \mid t' \;\to \langle \mathsf{B}, 1\ddot{\mathbb{B}} \rangle_y \mid \{\mathsf{A} : a\#N\} \mid t'$$

The first step lets the time pass, by rule [Delay]. In the second step, B fires the prefix `withdraw` B within the `after`, and in this way he collects $1\ddot{\mathbb{B}}$. This is obtained by using rule [C-Withdraw] in the premise of [C-Timeout].

We compile the timed commitment $\{G\}TC$, where for notational convenience we rewrite it as follows:

$$\begin{aligned}
TC &= D_1 + D_2\\
D_1 &= \mathtt{reveal}\; a.\, \mathtt{withdraw}\ \mathsf{A}\\
D_2 &= \mathtt{after}\; t : \mathtt{withdraw}\ \mathsf{B}
\end{aligned}$$

Assume that: $\mathsf{A} \in \mathsf{Hon}$, $\mathbf{txout}(x) = (\mathsf{T}_\mathsf{A}, 0)$ for some T_A whose output 0 has value $1\ddot{\mathbb{B}}$ and is redeemable by A, $\mathbf{sechash}(a) = h_a$, $\mathbf{val}(x) = 1\ddot{\mathbb{B}}$. Similarly, for B: $\mathbf{txout}(y) = (\mathsf{T}_\mathsf{B}, 0)$, and $\mathbf{val}(y) = 0\ddot{\mathbb{B}}$.

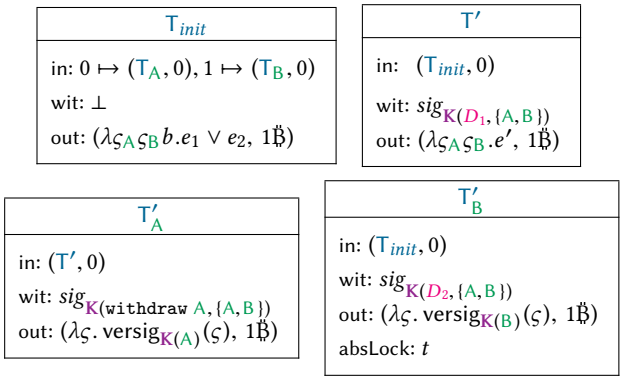

Figure 5: Transactions obtained by compiling the timed commitment contract.

The compiler produces $\ddot{\mathbb{B}}_{\mathbf{adv}}(\{G\}TC) = \mathsf{T}_{init}\mathcal{T}_1\mathcal{T}_2$, where:

$$\begin{aligned}
\mathcal{T}_1 &= \ddot{\mathbb{B}}_\mathsf{D}(D_1, D_1, \mathsf{T}_{init}, 0, 1\ddot{\mathbb{B}}, \{\mathsf{A}, \mathsf{B}\}, 0)\\
&= \ddot{\mathbb{B}}_\mathsf{C}(\mathtt{withdraw}\ \mathsf{A}, D_1, \mathsf{T}_{init}, 0, 1\ddot{\mathbb{B}}, \emptyset, \{\mathsf{A}, \mathsf{B}\}, 0)\\
&= \mathsf{T}'\ddot{\mathbb{B}}_\mathsf{D}(\mathtt{withdraw}\ \mathsf{A}, \mathtt{withdraw}\ \mathsf{A}, \mathsf{T}', 0, 1\ddot{\mathbb{B}}, \{\mathsf{A}, \mathsf{B}\}, 0) = \mathsf{T}'\mathsf{T}'_\mathsf{A}\\
\mathcal{T}_2 &= \ddot{\mathbb{B}}_\mathsf{D}(D_2, D_2, \mathsf{T}_{init}, 0, 1\ddot{\mathbb{B}}, \{\mathsf{A}, \mathsf{B}\}, 0)\\
&= \ddot{\mathbb{B}}_\mathsf{D}(\mathtt{withdraw}\ \mathsf{B}, D_2, \mathsf{T}_{init}, 0, 1\ddot{\mathbb{B}}, \{\mathsf{A}, \mathsf{B}\}, t) = \mathsf{T}'_\mathsf{B}
\end{aligned}$$

The obtained transactions are in Figure 5, where:

$$\begin{aligned}
e_1 &= \ddot{\mathbb{B}}_{\mathbf{out}}(D_1)\\
&\quad = \mathrm{versig}_{\mathsf{K}(D_1,\{\mathsf{A},\mathsf{B}\})}(\varsigma_\mathsf{A}\varsigma_\mathsf{B}) \;\wedge\; \mathsf{H}(b) = h_a \;\wedge\; |b| \geq \eta\\
e_2 &= \ddot{\mathbb{B}}_{\mathbf{out}}(D_2) = \mathrm{versig}_{\mathsf{K}(D_2,\{\mathsf{A},\mathsf{B}\})}(\varsigma_\mathsf{A}\varsigma_\mathsf{B})\\
e' &= \ddot{\mathbb{B}}_{\mathbf{out}}(\mathtt{withdraw}\ \mathsf{A}) = \mathrm{versig}_{\mathsf{K}(\mathtt{withdraw}\ \mathsf{A},\{\mathsf{A},\mathsf{B}\})}(\varsigma_\mathsf{A}\varsigma_\mathsf{B})
\end{aligned}$$

For the sake of readability we do not use distinct variables for different signatures of the same participant. The participants start by generating the transactions off chain, and exchanging the signatures shown in Figure 5. Doing this, A commits to her secret, whose hash h_a occurs in the output script T_{init}.out. After that, both A and B sign T_{init}, and put it on the ledger, stipulating the contract. The transaction T_{init} can be redeemed either by T' or by T'_B. In the first case, T' has to add to her witness a value a such that $H(a) = h_a$ and $|a| \geq \eta$. After that, A can redeem her deposit (now in T') by putting T'_A on the blockchain. In the second case, T_{init} can be redeemed by T'_B: however, this transaction can be put on the blockchain only after t, because of the timelock in T'_B.