Verifying Liquidity of Bitcoin Contracts

Massimo Bartoletti[1(✉)] and Roberto Zunino[2]

[1] Università degli Studi di Cagliari, Cagliari, Italy
bart@unica.it
[2] Università degli Studi di Trento, Trento, Italy

Abstract. A landmark security property of smart contracts is *liquidity*: in a non-liquid contract, it may happen that some funds remain frozen. The relevance of this issue is witnessed by a recent liquidity attack to the Ethereum Parity Wallet, which has frozen $\sim 160M$ USD within the contract, making this sum unredeemable by any user. We address the problem of verifying liquidity of Bitcoin contracts. Focussing on BitML, a contracts DSL with a computationally sound compiler to Bitcoin, we study various notions of liquidity. Our main result is that liquidity of BitML contracts is decidable, in all the proposed variants. To prove this, we first transform the infinite-state semantics of BitML into a finite-state one, which focusses on the behaviour of any given set of contracts, abstracting the context moves. With respect to the chosen contracts, this abstraction is sound and complete. Our decision procedure for liquidity is then based on model-checking the finite space of states of the abstraction.

Keywords: Bitcoin · Smart contracts · Verification

1 Introduction

Decentralized ledgers like Bitcoin and Ethereum [19,32] enable the trustworthy execution of *smart contracts*—computer protocols which regulate the exchange of assets among mutually untrusted users. The underlying protocols used to update the ledger (which defines the state of each contract) ensure that, even without trusted intermediaries, the execution of contracts is correct with respect to the contract rules. However, it may happen that the rules themselves are not correct with respect to the behaviour expected by the users. Indeed, all the attacks to smart contracts successfully carried out so far, which have plundered or frozen millions of USD in Ethereum [1–3,8,27,30], exploit some discrepancy between the intended and the actual behaviour of a contract.

To counteract these attacks, the research community has recently started to formalize smart contracts and their security properties [22–24], and to develop automated verification tools based on these models [21,27,31,35]. As a matter of fact, most of this research is targeted to Ethereum, the most widespread (and attacked) platform for smart contracts: for this reason, the security properties addressed by current tools focus on specific features of Solidity, the high-level language for smart contracts in Ethereum. For instance, some vulnerability patterns

F. Nielson and D. Sands (Eds.): POST 2019, LNCS 11426, pp. 222–247, 2019.
https://doi.org/10.1007/978-3-030-17138-4_10

checked by these tools are reentrancy and mishandled exceptions, whose peculiar implementation in Solidity has led to attacks, like to one to the DAO [1]. Only a few tools verify *general* security properties of smart contracts, that would be meaningful also outside the realm of Ethereum. Among these works, [35] checks a property called *liquidity*, which holds when the contract always admits a trace where its balance is decreased (so, the funds stored within the contract do not remain frozen). This has been inspired from a recent attack to Ethereum [2], which has frozen ∼160M USD within a contract, exploiting a bug in a library. While being capable of classifying this particular contract as non-liquid, any contract where the adversary can lock some funds and redeem them at a later moment would be classified as liquid. Stronger notions of liquidity may rule out these unsafe contracts, e.g. by checking that funds are never frozen *for all* possible strategies of the adversary. Studying liquidity in a more general setting would be important for various reasons. First, taking into account adversaries would allow to detect more security issues w.r.t. those checked by the current verification tools. Second, platform-agnostic notions of liquidity could be applied to the forthcoming blockchain technologies, e.g. [20,34]. Third, studying liquidity in simpler settings than Ethereum could simplify the verification problem, which is undecidable in Turing-powerful languages like those supported by Ethereum.

Contributions. We study several notions of liquidity for smart contracts, in a general setting where their behaviour is defined as a transition system. We then consider the special case where contracts are expressed in BitML, a high-level DSL for smart contracts which compiles into Bitcoin [14]. In such setting, we develop a verification technique for liquidity of smart contracts. We can summarise our main contributions as follows:

1. We formalize a notion of liquidity (Definition 2), and we illustrate several meaningful variants. Our notion of liquidity takes into account both the contract and the *strategy* that a participant follows to perform contract actions. Roughly, a strategy is liquid when following it ensures that funds do not remain frozen within the contract, even in the presence of adversaries.
2. We introduce an abstraction of the semantics of BitML which is finite-state (Theorem 1), and sound and complete w.r.t. the concrete (infinite-state) semantics, given a set of contracts under observation (Theorems 2 and 3).
3. We devise a verification technique for liquidity in BitML. Our technique can establish whether a strategy is liquid for a given contract, and also to synthesise a liquid strategy, when it exists (Theorem 4).

Our finite-state abstraction is general-purpose: verifying liquidity is only one of its possible applications (some other applications are discussed in Sect. 6).

Related Works. Several recent works study security issues related to Ethereum smart contracts. A few papers address EVM, the bytecode language which is the target of compilation of Solidity. Among them, [27] introduces an operational semantics of a simplified version of EVM, and develops Oyente, a tool to detect some vulnerability patterns of EVM contracts through symbolic execution. Securify [35] checks vulnerability patterns by analysing dependency graphs extracted

from EVM code. As mentioned before, this tool also addresses a form of liquidity, which essentially assumes a cooperating adversary. EtherTrust [21] is a framework for the static verification of EVM contracts, which can establish e.g. the absence of reentrancy vulnerabilities. This tool is based on the detailed formalisation of EVM provided in [22], which is validated against the official Ethereum test suite. The work [23] introduces an executable semantics of EVM, specified in the $\mathbb{K}$ framework. The tool in [18] translates Solidity and EVM code into F^*, and use its verification tools to detect vulnerabilities of contracts; further, the tool verifies the equivalence between a Solidity program and an alleged compilation of it into EVM. The work [24] verifies EVM code through the Isabelle/HOL proof assistant [33], proving that, upon an invocation of a specific contract, only its owner can decrease the balance.

Smart contracts in Bitcoin have a completely different flavour compared to Ethereum, since they are usually expressed as cryptographic protocols, rather than as programs. Despite the limited expressiveness of the scripts in Bitcoin transactions [10], several kinds of contracts for Bitcoin have been proposed [9]: they range from lotteries [6,7,13,29], to general multiparty computations [4,17,26], to contingent payments [11,28], etc. All these works focus on proving the security of a *fixed* contract, unlike the above-mentioned works on Ethereum, where the goal is to verify arbitrary contracts. As far as we know, only a couple of works pursue this goal for Bitcoin. The tool in [25] analyses Bitcoin scripts, in order to find under which conditions the enclosing transaction can be redeemed. Compared to [25], our work verifies contracts spanning among many transactions, rather than single scripts. The work [5] models contracts as timed automata, and then uses the Uppaal model checker [16] to verify their properties. The contracts modelled as in [5] cannot be directly translated to Bitcoin, while in our approach we can exploit the BitML compiler to translate contracts to standard Bitcoin transactions. Note also that the properties considered in [5] are specific to the modelled contract, while in this work we are interested in verifying general properties of contracts, like liquidity.

2 Overview

In this section we briefly overview BitML; we then give some intuition about liquidity and our verification technique. Because of space limits, we refer to [14] for a detailed treatment of BitML, and to [12] for a more gentle introduction.

We assume a set of *participants*, ranged over by $\mathsf{A}, \mathsf{B}, \ldots$, and a set of names, of two kinds: $x, y, \ldots$ denote *deposits* of ₿, while $a, b, \ldots$ denote *secrets*. We write $\boldsymbol{x}$ (resp. $\boldsymbol{a}$) for a finite sequence of deposit (resp. secrets) names.

2.1 BitML in a Nutshell

BitML is a domain-specific language for Bitcoin smart contracts, which allows participants to exchange cryptocurrency according to pre-agreed contract rules. In BitML, any participant can broadcast a *contract advertisement* $\{G\}C$, where

$G ::=$	precondition	$D ::=$	guarded contract
$\mathtt{A}:!\,v\,@\,x$	persistent deposit	$\mathtt{withdraw}\ \mathtt{A}$	transfer balance to A
$\mid \mathtt{A}:?\,v\,@\,x$	volatile deposit	$\mid \mathtt{split}\ \boldsymbol{v} \to \boldsymbol{C}$	split balance ($\lvert\boldsymbol{v}\rvert = \lvert\boldsymbol{C}\rvert$)
$\mid \mathtt{A}:\mathtt{secret}\,a$	committed secret	$\mid \mathtt{A} : D$	wait A's authorization
$\mid G \mid G'$	composition	$\mid \mathtt{after}\,t : D$	wait until time t
$C ::= \sum_{i\in I} D_i$	contract	$\mid \mathtt{put}\,\boldsymbol{x}\,\&\,\mathtt{reveal}\,\boldsymbol{a}\,\mathtt{if}\,p.\,C$	collect deposits/secrets

Fig. 1. Syntax of BitML contracts and preconditions.

$p ::=$	predicate	$E ::=$	expression
true	truth	N	32-bit constant
$\mid p \wedge p$	conjunction	$\mid \lvert a\rvert$	length of a secret
$\mid \neg p$	negation	$\mid E \circ E$	$(\circ \in \{+,-\})$
$\mid E \circ E$	$(\circ \in \{=,<\})$		

Fig. 2. Syntax of predicates.

C is the actual contract, specifying the rules to transfer bitcoins (฿), while G is a set of *preconditions* to its execution.

Preconditions (Fig. 1, left) may require participants to deposit some ฿ in the contract (either upfront or at runtime), or to commit to some secret. More in detail, $\mathtt{A}:!\,v\,@\,x$ requires A to own v฿ in a deposit x, and to spend it for stipulating a contract C. Instead, $\mathtt{A}:?\,v\,@\,x$ only requires A to pre-authorize the spending of x, which can be gathered by the contract at run-time. The precondition $\mathtt{A}:\mathtt{secret}\,a$ requires A to commit to a secret a before C starts.

After $\{G\}C$ has been advertised, each participant can choose whether to accept it, or not. When all the preconditions G have been satisfied, and all the involved participants have accepted, the contract C becomes *stipulated*. The contract starts its execution with a balance, initially set to the sum of the `!`-deposits required by its preconditions. Running C will affect this balance, when participants deposit/withdraw funds to/from the contract.

A contract C is a *choice* among zero or more branches. Each branch is a *guarded contract* (Fig. 1, right) which enables an action, and possibly proceeds with a continuation C'. The guarded contract `withdraw` A transfers the whole balance to A, while $\mathtt{split}\ v_1 \to C_1 \mid \cdots \mid v_n \to C_n$ decomposes the contract into n parallel components C_i, each one with balance v_i. The guarded contract $\mathtt{put}\,\boldsymbol{x}\,\&\,\mathtt{reveal}\,\boldsymbol{a}\,\mathtt{if}\,p$ atomically performs the following: (i) spend all the `?`-deposits $\boldsymbol{x}$, adding their values to the contract balance; (ii) check that all the secrets $\boldsymbol{a}$ have been revealed and satisfy the predicate p (Fig. 2). When enabled, the above-mentioned actions can be fired by anyone, at anytime. To restrict *who* can execute actions and *when*, one can use the decoration $\mathtt{A} : D$, which requires the authorization of A, and the decoration $\mathtt{after}\,t : D$, which requires to wait until time t.

A Basic Example. As a first example, we express in BitML the *timed commitment* [6], a basic protocol to construct more complex contracts, like e.g. lotteries and other games [7]. In the timed commitment, a participant A wants to choose a secret, and promises to reveal it before some time t. The contract ensures that if A does not reveal the secret in time, then she will pay a penalty of 1฿ to B (e.g., the opponent player in a game). In BitML, this is modelled as follows:

$$\{\mathsf{A} : ! \, 1 \, @ \, x \mid \mathsf{A} : \texttt{secret}\, a\}\ (\texttt{reveal}\, a.\, \texttt{withdraw}\ \mathsf{A} \ \ + \ \ \texttt{after}\, t : \texttt{withdraw}\ \mathsf{B})$$

The precondition requires A to pay upfront 1฿, and to commit to a secret a. The contract (hereafter, named TC) is a non-deterministic choice between two branches. Only A can choose the first branch, by performing `reveal` a (syntactic sugar for `put` [] & `reveal` a `if` *true*). Subsequently, anyone can transfer 1฿ to A. Only after t, if the `reveal` has not been fired, any participant can fire `withdraw` B in the second branch, moving 1฿ to B. So, before t, A has the option to reveal a (avoiding the penalty), or to keep it secret (paying the penalty). If no branch is taken by t, the first one who fires its `withdraw` gets 1฿.

2.2 BitML Semantics

We briefly recall from [14] the semantics of BitML. The semantics is a labelled transition system between configurations of the following form:

- $\{G\}C$, representing the advertisement of contract C with preconditions G;
- $\langle C, v\rangle_x$, representing a stipulated contract, holding a current balance of v฿. The name x uniquely identifies the contract in a configuration;
- $\langle \mathsf{A}, v\rangle_x$ representing a fund of v฿ owned by A, and with unique name x;
- $\mathsf{A}[\chi]$, representing A's *authorizations* to perform some operation χ. We refer to [14] for the syntax of authorizations (some of them are exemplified below);
- $\{\mathsf{A} : a\#N\}$, representing that A has committed to a random secret a with (secret) length N;
- $\mathsf{A} : a\#N$, representing that A has revealed her secret a (with its length N).
- $\Gamma \mid \Delta$ is the parallel composition of two configurations (with identity 0);
- $\Gamma \mid t$ is a *timed* configuration, where $t \in \mathbb{N}$ is a global time.

We now illustrate the BitML semantics by examples; when time is immaterial, we only show the steps of the untimed semantics. We omit labels on transitions.

Deposits. When A owns a deposit $\langle \mathsf{A}, v\rangle_x$, she can use it in various ways: 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 donate a deposit x to B, A must first issue the authorization $\mathsf{A}[x \triangleright \mathsf{B}]$; then, anyone can transfer the money to B:

$$\langle \mathsf{A}, v\rangle_x \mid \cdots \ \to \ \langle \mathsf{A}, v\rangle_x \mid \mathsf{A}[x \triangleright \mathsf{B}] \mid \cdots \ \to \ \langle \mathsf{B}, v\rangle_y \mid \cdots \qquad (y \text{ fresh})$$

We assume that whenever a participant authorizes an operation on some deposit x, then she is also authorising a self-donation $\mathsf{A}[x \triangleright \mathsf{A}]$ of such deposit.[1]

[1] This assumption, while helpful to simplify the subsequent technical development, does not allow an adversary to steal money; at worst, the adversary can use the authorization to transfer the money back to the original owner.

Advertisement. Any participant can advertise a new contract C (with preconditions G). This is obtained by performing the step $\Gamma \to \Gamma \mid \{G\}C$.

Stipulation. Stipulation turns a contract advertisement into an active contract. For instance, let $G = \mathsf{A} : ! \, 1 \, @ \, x \mid \mathsf{A} : ? \, 1 \, @ \, y \mid \mathsf{A} : \mathtt{secret} \, a$. Given a contract C, the stipulation of $\{G\}C$ is done in a few steps:

$$\langle \mathsf{A}, 1 \rangle_x \mid \langle \mathsf{A}, 1 \rangle_y \mid \{G\}C \;\to^*\; \langle \mathsf{A}, 1 \rangle_y \mid \langle C, 1 \rangle_z \mid \{\mathsf{A} : a \# N\}$$

Above, the funds in the deposit x are transferred to the newly created contract, to fulfill the precondition $\mathsf{A} : ! \, 1 \, @ \, x$. Instead, the deposit y remains in the configuration, to be possibly spent after some time. The component $\{\mathsf{A} : a \# N\}$ represents the secret committed to by A, with its length N.

Withdraw. Executing `withdraw` A terminates the contract, and transfers its whole balance to A by creating a fresh deposit owned by A:

$$\langle \mathtt{withdraw} \; \mathsf{A} + C', v \rangle_x \;\to\; \langle \mathsf{A}, v \rangle_y \qquad (y \text{ fresh})$$

Above, `withdraw` A is executed as a branch within a choice: as usual, taking a branch discards the other ones (denoted as C').

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

$$\langle (\mathtt{split} \; v_1 \to C_1 \mid v_2 \to C_2), v_1 + v_2 \rangle_x \to \langle C_1, v_1 \rangle_y \mid \langle C_2, v_2 \rangle_z \qquad (y, z \text{ fresh})$$

Put & Reveal. A prefix $\mathtt{put} \; z \; \& \; \mathtt{reveal} \; a \; \mathtt{if} \; p$ can be fired when the previously committed secret a (satisfying the predicate p) has been revealed, and the deposit z is available in the configuration. For instance:

$$\begin{aligned} & \langle \mathtt{put} \; z \; \& \; \mathtt{reveal} \; a \; \mathtt{if} \; |a| = N . \, C, v \rangle_x \mid \langle \mathsf{A}, v' \rangle_z \mid \{\mathsf{A} : a \# N\} \\ \to \; & \langle \mathtt{put} \; z \; \& \; \mathtt{reveal} \; a \; \mathtt{if} \; |a| = N . \, C, v \rangle_x \mid \langle \mathsf{A}, v' \rangle_z \mid \mathsf{A} : a \# N \\ \to \; & \langle C, v + v' \rangle_y \mid \mathsf{A} : a \# N \end{aligned}$$

In the first step, A reveals her secret a. In the second step, any participant fires the prefix; doing so rakes the deposit z within the contract.

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

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

In the first step, A authorizes to take the branch `withdraw` B. After that, any participant can fire such branch.

Time. We always allow time t to advance by a delay $\delta > 0$, through a transition $\Gamma \mid t \to \Gamma \mid t + \delta$. Advancing time can enable branches decorated with $\texttt{after}\, t$. For instance, if $t_0 + \delta \geq t$, we have the following computation:

$$\begin{aligned} &\langle (\texttt{after}\, t : \texttt{withdraw}\, \mathsf{B}) + C', v\rangle_x \mid t_0 \\ \to\ &\langle (\texttt{after}\, t : \texttt{withdraw}\, \mathsf{B}) + C', v\rangle_x \mid t_0 + \delta \ \to\ \langle \mathsf{B}, v\rangle_y \mid t_0 + \delta \end{aligned}$$

Runs and Strategies. A *run* $\mathcal{R}$ is a (possibly infinite) sequence:

$$\Gamma_0 \mid t_0 \xrightarrow{\ell_0} \Gamma_1 \mid t_1 \xrightarrow{\ell_1} \cdots$$

where ℓ_i are the transition labels, Γ_0 contains only deposits, and $t_0 = 0$. If $\mathcal{R}$ is finite, we write $\Gamma_\mathcal{R}$ for its last untimed configuration, and $\delta_\mathcal{R}$ for its last time. A *strategy* Σ_A is a PPTIME algorithm which allows A to select which actions to perform (possibly, time delays), among those permitted by the BitML semantics. The choice among these actions is controlled by the adversary strategy Σ_{Adv}, which acts on behalf of all the dishonest participants. Given the strategies of all participants (including Adv), there is a unique run *conforming* to all of them.

2.3 Liquidity

A desirable property of smart contracts is *liquidity*, which requires that the contract balance is always eventually transferred to some participant. In a non-liquid contract, funds can be frozen forever, unavailable to anyone, hence effectively destroyed. There are many possible flavours of liquidity, depending e.g. on which participants are assumed to be honest, and on which are their strategies. The simplest form of liquidity is to consider the case where everyone cooperates: i.e. a contract is liquid if there exists some strategy for each participant such that no funds are ever frozen. However, this notion does not capture the essence of smart contracts, i.e. to allow mutually untrusted participants to safely interact.

For instance, consider the following contract, where A and B contribute $1\ddot{\mathrm{B}}$ each for a donation of $2\ddot{\mathrm{B}}$ to either C or D (we omit the preconditions for brevity):

$$\mathsf{A} : \mathsf{B} : \texttt{withdraw}\, \mathsf{C} \ + \ \mathsf{A} : \mathsf{B} : \texttt{withdraw}\, \mathsf{D}$$

In order to unlock the funds, A and B must agree on the recipient of the donation, by giving their authorization on the same branch. This contract would be liquid only by assuming the cooperation between A and B: indeed, A alone cannot guarantee that the $2\ddot{\mathrm{B}}$ will eventually be donated, as B can choose a different recipient, or even refuse to give any authorization. Consequently, unless A trusts B, it makes sense to consider this contract as non-liquid, from the point of view of A (and for similar reasons, also from that of B).

Consider now the timed commitment contract discussed before:

$$\texttt{reveal}\, a.\, \texttt{withdraw}\, \mathsf{A} \ + \ \texttt{after}\, t : \texttt{withdraw}\, \mathsf{B}$$

This contract is liquid from A's point of view (even if B is dishonest), because A can reveal the secret and then redeem the funds from the contract. The timed commitment is also liquid from B's point of view: if A does not reveal the secret (making the first branch stuck), the funds in the contract can be redeemed through the second branch, after time t.

In a *mutual* timed commitment contract, where A and B have to exchange their secrets or pay a 1₿ penalty, achieving liquidity is a bit more challenging. We first consider a wrong attempt:

$$\begin{aligned}&\texttt{reveal}\, a.\texttt{reveal}\, b.\texttt{split}\ (1฿ \to \texttt{withdraw}\ \mathsf{A} \mid 1฿ \to \texttt{withdraw}\ \mathsf{B})\\ &+ \texttt{after}\, t : \texttt{withdraw}\ \mathsf{B}\end{aligned}$$

Intuitively, A has only the following strategies, according to when she decides to reveal her secret a: (i) A chooses to reveal a unconditionally, and to perform the `reveal` a action. This strategy is *not* liquid: indeed, if B does not reveal b, the contract is stuck. (ii) A chooses to reveal a only *after* B has revealed b. This strategy is *not* liquid: indeed, if B chooses not to reveal b, the contract will never advance. (iii) A chooses to wait until B reveals secret b, or until time $t' \geq t$, whichever comes first. If b was revealed, A reveals a, and splits the contract balance between A and B. Otherwise, if the deadline t' is expired, A transfers the whole balance to B. Note that, although this strategy is liquid, it is not satisfactory for A, since in the second case she will lose money.

This example highlights a crucial point: participants' strategies have to be taken into account when defining liquidity. Indeed, the mere fact that a liquid strategy exists does not imply that it is the ideal strategy for the honest participant. To fix this issue, we revise the mutual timed commitment as follows:

$$\begin{aligned}&\texttt{reveal}\, a.\,\big(\texttt{reveal}\, b.\texttt{split}\ (1฿ \to \texttt{withdraw}\ \mathsf{A} \mid 1฿ \to \texttt{withdraw}\ \mathsf{B})\\ &\qquad + \texttt{after}\, t' : \texttt{withdraw}\ \mathsf{A}\big)\\ &+ \texttt{after}\, t : \texttt{withdraw}\ \mathsf{B}\end{aligned}$$

where $t < t'$. Now, A has a liquid strategy where she does not pay the penalty. First, A reveals a before time t. After that, if B reveals b, then A can execute the `split`, transferring 1₿ to herself and 1₿ to B (note that this does not require B's cooperation); otherwise, after time t', A can withdraw 2₿ by executing the `withdraw` A in the `after` $t' : \cdots$ branch.

These examples, albeit elementary, show that detecting if a strategy is liquid for a contract is not straightforward, in general. The problem of determining a liquid strategy for a given contract seems even more demanding. Automatic techniques for the verification and inference of liquid strategies can be useful tools for the developers of smart contracts.

2.4 Verifying Liquidity

One of the main contributions of this paper is a verification technique for the liquidity of BitML contracts. Our technique is based on a more general result,

i.e. a strict correspondence between the semantics of BitML in [14] (hereafter, called *concrete* semantics) and a new abstract semantics, which is finite-state (Theorem 1). Our abstraction is a correct and complete approximation of the concrete semantics with respect to a given set of contracts (Theorems 2 and 3). To obtain a finite-state abstraction, we need to cope with three sources of infiniteness of the concrete semantics of BitML: the unbounded passing of time, the advertisement/stipulation of new contracts, and the operations on deposits. Our abstraction replaces the time t in concrete configurations with a finite number of time intervals $T = [t_0, t_1)$, and it disables the transitions to advertise new contracts. Further, the only operations on deposits allowed by the abstract semantics are the ones for transferring them to contracts and for destroying them. The latter is needed e.g. to properly model the situation where a participant spends a `?`-deposit.

The intended use of our abstraction is to start from a configuration containing an arbitrary (but finite) set of contracts, and then analyse their possible evolutions in the presence of an honest participant and an adversary. This produces a finite set of (finite) traces, which we can model-check for liquidity. Soundness and completeness of the abstraction are exploited to prove that liquidity is decidable (Theorem 4). The computational soundness of the BitML compiler [14] guarantees that if a contract is verified to be liquid according to our analysis, this property is preserved when executing it on Bitcoin.

3 Liquidity

In this section we formalise a notion of liquidity of contracts, and we suggest some possible variants. Aiming at generality, liquidity is parameterised over (i) a set X of contract names, uniquely identifying the contracts under observation; (ii) a participant A (with her strategy Σ_{A}), which we assume to be the only honest participant in the system. Roughly, we want that the funds stored within the contracts X are eventually transferred to some participant, in any run conforming to A's strategy. The actual definition is a bit more complex, because the other participants may play against A, e.g. avoiding to reveal their secrets, or to give their authorizations for some branch.

We start by introducing an auxiliary partial function $orig_{\mathcal{R}_0}(\mathcal{R}, x)$ that, given a contract name x and an extension $\mathcal{R}$ of a run $\mathcal{R}_0$, determines the ancestor y of x in the last configuration of $\mathcal{R}_0$, if any. Intuitively, $orig_{\mathcal{R}_0}(\mathcal{R}, x) = y$ means that y has evolved into $\mathcal{R}$, eventually leading to x (and possibly to other contracts).

In BitML, there are only two ways to make a contract evolve into another contract. First, a `split` can spawn new contracts, e.g.:

$$\langle \texttt{split}\ (v_1 \to C_1 \mid v_2 \to C_2), v_1 + v_2 \rangle_x \xrightarrow{split(x)} \langle C_1, v_1 \rangle_{y_1} \mid \langle C_2, v_2 \rangle_{y_2}$$

Here, both y_1 and y_2 have x as ancestor. Second, `put`&`reveal` reduces as follows:

$$\langle \texttt{put}\ z\ \&\ \texttt{reveal}\ a.\, C, v \rangle_x \mid \langle \mathsf{A}, v' \rangle_z \mid \cdots \xrightarrow{put(z,a,x)} \langle C, v + v' \rangle_y \mid \cdots$$

In this case, the ancestor of y is x.

$$orig_{\mathcal{R}_0}(\mathcal{R}_0, x) = x \quad \text{if } x \in \mathrm{cn}(\Gamma_{\mathcal{R}_0})$$

$$orig_{\mathcal{R}_0}(\mathcal{R}' \xrightarrow{\ell} \Gamma, x) = \begin{cases} orig_{\mathcal{R}_0}(\mathcal{R}', x) & \text{if } x \in \mathrm{cn}(\mathcal{R}') \\ orig_{\mathcal{R}_0}(\mathcal{R}', y) & \text{if } \begin{array}{l} x \in \mathrm{cn}(\mathcal{R}' \xrightarrow{\ell} \Gamma) \setminus \mathrm{cn}(\mathcal{R}') \text{ and} \\ (\ell = split(y) \text{ or } \ell = put(\boldsymbol{z}, \boldsymbol{a}, y)) \end{array} \end{cases}$$

Fig. 3. Origin of a contract name within a run.

Definition 1. *Let $\mathcal{R}$ be a run extending some run $\mathcal{R}_0$, and let x be a contract name. We define $orig_{\mathcal{R}_0}(\mathcal{R}, x)$ by induction on the length of $\mathcal{R}$ in Fig. 3, where $\mathrm{cn}(\Gamma)$ denotes the set of contract names in Γ.*

Example 1. Let $\mathcal{R}_0$ be a run with last configuration $\Gamma_{\mathcal{R}_0} = \langle C_1, v\rangle_y \mid \langle \mathsf{A}, v\rangle_z$, and let $\mathcal{R}$ be the following extension of $\mathcal{R}_0$, where the contracts C_1 and C_2 are immaterial, but for the fact that they enable the displayed moves:

$$\begin{aligned} \langle C_1, v\rangle_y \mid \langle \mathsf{A}, v\rangle_z &\to \langle C_1, v\rangle_y \mid \langle \mathsf{A}, v\rangle_z \mid \{G\}C_2 \to^* \langle C_1, v\rangle_y \mid \langle C_2, v\rangle_x \\ &\xrightarrow{split(x)} \langle C_1, v\rangle_y \mid \langle C_2', v\rangle_{x'} \\ &\xrightarrow{split(y)} \langle C_1', v'\rangle_{y'} \mid \langle C_1'', v - v'\rangle_{y''} \mid \langle C_2', v\rangle_{x'} \end{aligned}$$

We have that $orig_{\mathcal{R}_0}(\mathcal{R}, y') = orig_{\mathcal{R}_0}(\mathcal{R}, y'') = y$, since the corresponding contracts have been obtained through a split of the ancestor y, which was in the last configuration of $\mathcal{R}_0$. Instead, $orig_{\mathcal{R}_0}(\mathcal{R}, x')$ is undefined, because its ancestor x is not in $\mathcal{R}_0$. Further, $orig_{\mathcal{R}_0}(\mathcal{R}, y) = y$, while $orig_{\mathcal{R}_0}(\mathcal{R}, x)$ is undefined.

We now formalise liquidity. Assume that we want to observe a single contract x, occurring in the last configuration of some run $\mathcal{R}_0$ (note that x has been stipulated at some point during $\mathcal{R}_0$). A participant A wants to know if the strategy Σ_{A} allows her to make x evolve so that funds are never frozen within the contract. We require that A can do this *without* the help of the other participants, which therefore we model as a single adversary Adv. More precisely, we say that x is liquid for A when, after any extension $\mathcal{R}$ of $\mathcal{R}_0$, Σ_{A} can choose a sequence of moves so to make all the descendant contracts of x terminate, transferring their funds to some participant (possibly not A). Note that such moves can not reveal secrets of other participants, or generate authorizations for them: A must be able to unfreeze the funds on her own, using her strategy. By contrast, $\mathcal{R}$ can also involve such moves, but it must conform to A's strategy. The actual definition of liquidity generalises the above to sets X_0 of contract names.

Definition 2 (Liquidity). *Let A be an honest participant, with strategy Σ_{A}, let $\mathcal{R}_0$ be a run, and let X_0 be a set of contract names in $\Gamma_{\mathcal{R}_0}$. We say that X_0 is* liquid w.r.t. *Σ_{A} in $\mathcal{R}_0$ if, for all finite extensions $\mathcal{R}$ of $\mathcal{R}_0$ conforming to Σ_{A} and to some Σ_{Adv}, there exists an extension $\mathcal{R}' = \mathcal{R} \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_n}$ of $\mathcal{R}$ such that:*

$$\forall i \in 1..n \ : \ \ell_i \in \Sigma_{\mathsf{A}}(\mathcal{R} \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_{i-1}}) \tag{1}$$

$$x \in \mathrm{cn}(\Gamma_{\mathcal{R}'}) \implies orig_{\mathcal{R}_0}(\mathcal{R}', x) \notin X_0 \tag{2}$$

Condition (1) requires that all the moves after $\mathcal{R}$ can be taken by A alone, conforming to her strategy. Condition (2) checks that $\mathcal{R}'$ no longer contains descendants of the contracts X_0: since in BitML active contracts always store some funds, this is actually equivalent to checking that funds are not frozen.

We remark that, although Definition 2 is instantiated on BitML, the basic concepts it relies upon (runs, strategies, termination of contracts) are quite general. Hence, our notion of liquidity, as well as the variants proposed below, can be applied to other languages for smart contracts, using their transition semantics.

Example 2. Recall the timed commitment contract TC from Sect. 2. Assume that A's strategy is to wait until time $t - 1$ (i.e., one time unit before the deadline), then reveal the secret and fire `withdraw` A. Let $\mathcal{R}_0$ be a run with final configuration $\langle TC, 1\ddot{\mathrm{B}}\rangle_x \mid \{\mathsf{A} : a\#N\}$, for some length N. We have that $\{x\}$ is liquid w.r.t. Σ_{A} in $\mathcal{R}_0$, while it is *not* liquid w.r.t. the strategy where A does not reveal the secret, or reveals it without firing `withdraw` A. Indeed, under these strategies A alone cannot make x terminate.

Example 3. Consider the following two contracts, which both require as precondition that A put a deposit of $2\ddot{\mathrm{B}}$ and commits to a secret a, and where p is an arbitrary predicate on a:

$$\begin{aligned} C_1 &= \texttt{reveal}\, a\, \texttt{if}\, p.\, \texttt{withdraw}\ \mathsf{A} + \texttt{reveal}\, a\, \texttt{if}\, \neg p.\, \texttt{withdraw}\ \mathsf{B} \\ C_2 &= \texttt{split}\ \ 1\ddot{\mathrm{B}} \to \texttt{reveal}\, a\, \texttt{if}\, p.\, \texttt{withdraw}\ \mathsf{A} \\ &\qquad\quad \mid 1\ddot{\mathrm{B}} \to \texttt{reveal}\, a\, \texttt{if}\, \neg p.\, \texttt{withdraw}\ \mathsf{B} \end{aligned}$$

Assume that A's strategy is to reveal the secret, and then fire any enabled `withdraw`. Under this strategy, C_1 is liquid, because one of the `reveal` branches is enabled, and the corresponding `withdraw` is fired, transferring $2\ddot{\mathrm{B}}$ either to A or to B. Instead, no strategy of A can make C_2 liquid. If A does not reveal the secret, then the $2\ddot{\mathrm{B}}$ are frozen; otherwise, if A reveals the secret, then only one of the two descendents of C_2 can fire the `reveal`, and so $1\ddot{\mathrm{B}}$ remains frozen.

Example 4 (Lottery). Consider a lottery between two players. The preconditions require A and B to commit to one secret each (a and b, respectively), and to put a deposit of $3\ddot{\mathrm{B}}$ each ($1\ddot{\mathrm{B}}$ as a bet, and $2\ddot{\mathrm{B}}$ as a penalty for dishonest behaviour):

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

The contract splits the balance in three parts, of $2\ddot{\mathrm{B}}$ each. The first part allows B to reveal b and then redeem $2\ddot{\mathrm{B}}$; otherwise, after the deadline A can redeem

B's penalty (as in the timed commitment). Similarly, the second part allows A to redeem 2₿ by revealing a. To determine the winner we compare the secrets, in the subcontract *Win*: A wins if the secrets have the same length, otherwise B wins. This lottery is *fair*, since: (i) if both players are honest, then they will reveal their secrets within the deadlines (redeeming 2₿ each), and then they will have a 1/2 probability of winning[2]; (ii) if a player is dishonest, not revealing the secret, then the other player has a positive payoff, since she can redeem 4₿.

Although fair, *Lottery*(*Win*) is non-liquid w.r.t. *any* strategy of A. Indeed, if B does not reveal his secret, then the 2₿ stored in the *Win* subcontract are frozen. We can recover liquidity by replacing *Win* with the following:

$$\begin{aligned} \mathit{Win}_2 = \mathit{Win} &+ (\texttt{after}\, t' : \texttt{reveal}\, a.\ \texttt{withdraw}\ \mathsf{A}) \\ &+ (\texttt{after}\, t' : \texttt{reveal}\, b.\ \texttt{withdraw}\ \mathsf{B}) \end{aligned}$$

where $t' > t$. In this case, even if B does not reveal b, A can use a strategy firing any enabled `withdraw` at time t', to unfreeze the 2₿ stored in Win_2.

We now present some variants of the notion of liquidity presented before.

Multiparty Liquidity. A straightforward generalisation of liquidity is to assume a set of honest participants (rather than just one). In this case, we can extend Definition 2 by requiring that the run $\mathcal{R}$ conforms to the strategies of all honest participants, and the moves in (1) can be taken by any honest participant.

We illustrate this notion through the following escrow contract between two participants A and B, where the precondition requires A to deposit 1₿:

$$\begin{aligned} \mathit{Escrow} &= \mathsf{A} : \texttt{withdraw}\ \mathsf{B} \ + \ \mathsf{B} : \texttt{withdraw}\ \mathsf{A} \ + \ \mathsf{A} : \mathit{Resolve} \ + \ \mathsf{B} : \mathit{Resolve} \\ \mathit{Resolve} &= \texttt{split}(0.1₿ \to \texttt{withdraw}\ \mathsf{M} \\ &\qquad \mid \ 0.9₿ \to \mathsf{M} : \texttt{withdraw}\ \mathsf{A} + \mathsf{M} : \texttt{withdraw}\ \mathsf{B}) \end{aligned}$$

After the contract has been stipulated, A can choose to pay B, by authorizing the first branch. Similarly, B can allow A to take her money back, by authorizing the second branch. If they do not agree, any of them can invoke a mediator M to resolve the dispute, invoking a *Resolve* branch. There, the 1₿ deposit is split in two parts: 0.1₿ go to the mediator, while 0.9₿ are assigned either to A and B, depending on M's choice.

Assuming that only A is honest, this contract does not admit any liquid strategy for A, according to Definition 2. This is because B can invoke the mediator, who can refuse to act, freezing the funds within the contract. Similarly, B alone has no liquid strategy, as well as M. Instead, *Escrow* admits a liquid multiparty strategy for any pair of honest participants. For instance, if A and M are honest, their strategies could be the following. A chooses whether to authorize

[2] Note that B could increase his probability to win the lottery by choosing a secret with length $N > 1$. However, doing so will make B lose his 2₿ deposit in the first part of `split`, and so B's *average* payoff would be negative.

the first branch or not; in the first case, she fires `withdraw` B; otherwise, if B gives his authorization within a certain deadline, then A withdraws 1₿; if not, after the deadline A invokes M. The strategy of M is to authorize some participant to redeem the 0.9₿, and to fire all the `withdraw` within *Resolve*.

Strategyless Liquidity. Another variant of liquidity can be obtained by inspecting only the contract, neglecting A's strategy. In this case, we consider the contract as liquid when there exists some strategy of A which satisfies the constraints in Definition 2. For instance, the contract B : `withdraw` A is non-liquid from A's point of view, according to this notion, while it would be liquid for B.

Quantitative Liquidity. Definition 2 requires that no funds remain frozen within the contract. However, in some cases A could accept the fact that a portion of the funds remain frozen, especially when these funds would be ideally assigned to other participants. Following this intuition, we could define a contract *v-liquid* w.r.t. Σ_A if at least v bitcoins are guaranteed to be redeemable. If the contract uses only `!`-deposits, the special case where v is the sum of all these deposits corresponds to the notion in Definition 2. For instance, *Lottery*(*Win*) from Example 4 is non-liquid for any strategy of A, but it is 4₿-liquid if A's strategy is to reveal her secret, and perform all the enabled `withdraw`. Instead, *Lottery*(Win_2) is 6₿-liquid, and then also liquid, under this strategy.

A refinement of this variant could require that at least v₿ are transferred to A, rather than to any participant. Under this notion, both *Lottery*(*Win*) and *Lottery*(Win_2) would be 2₿-liquid for A. Further, *Lottery*(Win_2) would be 4₿-liquid in case A wins the lottery.

Liquidity with Unknown Secrets. All the notions of liquidity proposed so far depend on the initial run $\mathcal{R}_0$, which contains the lengths of the committed secrets. For instance, consider the run ending with the following configuration:

$$\{\mathsf{B} : b\#0\} \mid \langle (\texttt{reveal}\, b\, \texttt{if}\, |b| = 1.\ \mathsf{B} : \texttt{withdraw}\ \mathsf{A}) + \texttt{withdraw}\ \mathsf{A}, 1₿ \rangle_x$$

Since the length of b is zero, the `reveal` branch cannot be taken, so A has a liquid strategy (e.g., fire the `withdraw` A). Instead, in an alternative initial run where B chooses a secret of length 1, A has no liquid strategy, since B can reveal the secret and then deny his authorization, freezing 1₿.

In practice, when A performs the liquidity analysis, she does not know the secrets of other participants. To be safe, A should use a worst-case analysis, which would regard the contract (`reveal` b `if` $|b| = 1$. B : `withdraw` A) + `withdraw` A as non-liquid. We can obtain such worst-case analysis by verifying liquidity (in the flavour of Definition 2) for all possible choices of the lengths of Adv's secrets. Although there is an infinite set of such lengths, each contract only checks a finite set of `if` conditions. Hence, the infinite set of lengths can be partitioned into a finite set of regions, which can be used as samples for the analysis. In this way, the basic liquidity analysis is performed a finite number of times.

Similar worst-case analyses can be obtained for all the other above-mentioned variants of liquidity. An average-case analysis can be obtained by assuming to

know the probability distribution of A's secrets lengths, partitioning secrets lengths like in the worst-case analysis.

Other Variants. Mixing multiparty and strategyless liquidity, we obtain the notion of liquidity used in [35], in the context of Ethereum smart contracts. This notion considers a contract liquid if there exists a collaborative strategy of all participants that never freezes funds. Other variants may take into account the time when funds become liquid, the payoff of strategies (e.g., ruling out irrational adversaries), or fairness issues. Note indeed that Definition 2 already assumes a sort of fairness, by effectively forbidding the adversary to interfere when the honest participant attempts to unfreeze some funds. Technically, this is implemented in item (1) of Definition 2, requiring that the moves $\ell_1 \dots \ell_n$ are performed atomically. Atomicity might be realistic in some settings, but not in others. For instance, in Ethereum a sequence $\ell_1 \dots \ell_n$ of method calls can be performed atomically: this requires to deploy a new contract with a suitable method which performs the calls $\ell_1 \dots \ell_n$ in sequence, and then to invoke it. BitML, instead, does not allow participants to perform an atomic sequence of moves: an honest participant could start to perform the sequence, but at some point in the middle the adversary interferes. To make the contract liquid, the honest participant must still have a way to unfreeze the funds from the contract. Of course, the adversary could interfere once again, and so on. This could lead to an infinite trace where each attempt by the honest player is hindered by the adversary. However, this is not an issue in BitML, for the following reason. Since the moves $\ell_1 \dots \ell_n$ make the contract terminate, we can safely assume that each of these moves makes the contract progress (as moves which do not affect the contract can be avoided). Since a BitML contract can not progress forever without terminating (and unfreezing its funds), the honest participant just needs to be able to make a step at a time (with possible interferences by the adversary, which may affect the choice of the next step). Defining liquidity beyond BitML and Ethereum may require to rule out unfair runs, where the adversary prevents honest participants to perform the needed sequences of moves.

4 A Finite-State Semantics of BitML

The concrete BitML semantics is infinite-state because participants can always create new contracts and deposits, and can advance the current time (a natural number). In this section we introduce an abstract semantics for BitML, which focuses on both these features so to reduce the state space to a finite one. More specifically, for a concrete configuration $\Gamma \mid t$:

- we abstract Γ as an *abstract configuration* $\alpha_X(\Gamma)$, where X is the (finite) set of contract names under observation. Roughly, $\alpha_X(\Gamma)$ represents only the part of Γ needed to run the contracts X, discarding the other parts;
- we abstract t as a time interval $\alpha_{\mathcal{T}}(t) = [t_0, t_1)$, where $t_0, t_1 \in \mathcal{T} \cup \{0, +\infty\}$. The parameter $\mathcal{T}$ is a finite set of naturals, which intuitively represents all the deadlines occurring in the contracts X.

$$\alpha_{X,Z}(\langle C, v\rangle_x) = \begin{cases} \langle C, v\rangle_x & \text{if } x \in X \\ 0 & \text{otherwise} \end{cases} \qquad \alpha_{X,Z}(\{\mathsf{A} : a\#N\}) = \begin{cases} \{\mathsf{A} : a\#N\} & \text{if } a \in Z \\ 0 & \text{otherwise} \end{cases}$$

$$\alpha_{X,Z}(\langle \mathsf{A}, v\rangle_x) = \begin{cases} \langle \mathsf{A}, v\rangle_x & \text{if } x \in Z \\ 0 & \text{otherwise} \end{cases} \qquad \alpha_{X,Z}(\mathsf{A} : a\#N) = \begin{cases} \mathsf{A} : a\#N & \text{if } a \in Z \\ 0 & \text{otherwise} \end{cases}$$

$$\alpha_{X,Z}(\mathsf{A}[\chi]) = \begin{cases} \mathsf{A}[\chi] & \text{if } \chi = x \rhd D \text{ and } x \in X \\ \mathsf{A}[x, 0 \rhd y^\star] & \text{if } \chi = x \rhd \mathsf{B} \text{ and } x \in Z \\ 0 & \text{otherwise} \end{cases}$$

$$\alpha_{X,Z}(\{G\}C) = 0 \qquad \alpha_{X,Z}(\Delta \mid \Delta') = \alpha_{X,Z}(\Delta) \mid \alpha_{X,Z}(\Delta')$$

Fig. 4. Abstraction of configurations.

We start by defining the abstraction of configurations.

Definition 3 (Abstraction of configurations). *We define the function $\alpha_{X,Z}$ on concrete configurations in Fig. 4, where $y^\star$ denotes a fixed name not present in any concrete configuration. We write $\alpha_X(\Gamma)$ for $\alpha_{X,\mathcal{N}(X,\Gamma)}(\Gamma)$, where:*

$$\mathcal{N}(X,\Gamma) = \{z \mid \exists x, C, v, \Gamma' : \Gamma = \langle C, v\rangle_x \mid \Gamma' \ \wedge\ x \in X \ \wedge\ z \in \mathrm{dn}(C) \cup \mathrm{sn}(C)\}$$

where we denote with $\mathrm{dn}(C)$ the set of deposit names in some `put` *within C, and with $\mathrm{sn}(C)$ the set of secrets names in some* `reveal` *within C.*

The abstraction removes from Γ all the deposits not in Z, all the (committed or revealed) secrets not in Z, and all the authorizations enabling branches of some contracts not in Z. All the other authorizations—but the deposit authorizations, which are handled in a special way—are removed. This is because, in the concrete semantics, deposits move into fresh ones which are no longer relevant for the contracts X. Note that if we precisely tracked such irrelevant deposits and their authorizations, our abstract semantics would become infinite-state. To cope with this issue, the abstract semantics will render deposit moves as "destroy" moves, removing the now irrelevant deposits from the configuration. As anticipated in Sect. 2.2, an authorization of a deposit move can only be performed after a "self-donate" authorization $\mathsf{A}[x \rhd \mathsf{A}]$, which lets A transfer the funds in x to another of her deposits. Our abstraction maps such $\mathsf{A}[x \rhd \mathsf{A}]$ into an "abstract destroy" authorization $\mathsf{A}[x, 0 \rhd y^\star]$. In this way, in abstract configurations, deposits can be destroyed when, in concrete configurations, they are no longer relevant.

The abstraction of time $\alpha_{\mathcal{T}}$ is parameterised over a finite set of naturals $\mathcal{T}$, which partitions $\mathbb{N}$ into a finite set of non-overlapping intervals[3]. Each time t is abstracted as $\alpha_{\mathcal{T}}(t)$, which is the unique interval containing t.

[3] A specific choice of $\mathcal{T}$, which considers all the deadlines in the contracts X under observation, is defined later on (Definition 8).

Definition 4 (Abstraction of time). *Let* $\mathcal{T} \in \wp_{fin}(\mathbb{N})$. *We define the function* $\alpha_{\mathcal{T}} : \mathbb{N} \to \wp(\mathbb{N})$ *as* $\alpha_{\mathcal{T}}(t) = [t_0, t_1)$ *where:*

$$t_0 = \max\left(\{t' \in \mathcal{T} \mid t' \leq t\} \cup \{0\}\right) \quad t_1 = \min\left(\{t' \in \mathcal{T} \mid t' > t_0\} \cup \{+\infty\}\right)$$

Lemma 1. *If* $\mathcal{T} \in \wp_{fin}(\mathbb{N})$, *then: (i)* $\forall t \in \mathbb{N} : t \in \alpha_{\mathcal{T}}(t)$; *(ii)* $\operatorname{ran} \alpha_{\mathcal{T}}$ *is finite.*

Abstract Semantics. We now describe the abstract semantics of BitML (the detailed formalisation is deferred to Definition 7 in Appendix A). An *abstract configuration* is a term of the form $\Gamma \mid T$, where Γ is a concrete untimed configuration, and $T \in \operatorname{ran} \alpha_{\mathcal{T}}$. We then define the relation $\to_\sharp$ between abstract configurations by differences w.r.t. the concrete relation $\to$:

1. the rule to advertise contracts is removed.
2. the rules for deposits are replaced by two rules, which authorize and perform the destroy of deposits. In these rules we use the fixed name $y^\star$, unlike the fresh names in the concrete semantics, so to avoid infinite branching.
3. the rule for delays is replaced by a new rule, which allows for transitions $\Gamma \mid T \xrightarrow{\delta}_\sharp \Gamma \mid T'$. The delay δ is the least positive integer which makes T (in the earliest moment) step to T', i.e. $\delta = \min T' - \min T$.
4. the rule for making a contract $\langle \texttt{withdraw}\ \mathsf{A}, v\rangle_x$ reduce to a deposit $\langle \mathsf{A}, v\rangle_y$ is replaced so that $\langle \texttt{withdraw}\ \mathsf{A}, v\rangle_x$ reduces to 0 (the empty configuration).
5. the rule for making branches $\texttt{after}\, t : D$ evolve is adapted to time intervals. The new rule requires that the current time interval T is later than t.

Abstract Runs. Given an arbitrary abstract configuration $\Gamma_0 \mid T_0$, an *abstract run* $\mathcal{R}^\sharp$ is a (possibly infinite) sequence $\Gamma_0 \mid T_0 \to_\sharp \Gamma_1 \mid T_1 \to_\sharp \cdots$. While concrete runs always start (at time 0) from configurations which contain only deposits, abstract runs can start from arbitrary configurations.

Abstract Strategies. An *abstract strategy* $\Sigma^\#_{\mathsf{A}}$ is a PPTIME algorithm which allows A to select which actions to perform, among those permitted by the abstract semantics. Conformance between abstract runs and strategies is defined similarly to the concrete case [14].

Concretisation of Strategies. Each abstract strategy $\Sigma^\#_{\mathsf{A}}$ can be transformed into a concrete strategy $\Sigma_{\mathsf{A}} = \gamma(\Sigma^\#_{\mathsf{A}})$ as follows. The transformation is parameterised over a concrete run $\mathcal{R}_0$ and a set of contract names $X_0 \subseteq \operatorname{cn}(\Gamma_{\mathcal{R}_0})$: intuitively, $\mathcal{R}_0$ is the concrete counterpart of the initial abstract configuration $\Gamma_0 \mid T_0$, and X_0 is the set of contracts under observation. The strategy Σ_{A} receives as input a concrete run $\mathcal{R}$, and it must output the next actions. If $\mathcal{R}$ is a prefix of $\mathcal{R}_0$, the next move is chosen as in $\mathcal{R}_0$. The case where $\mathcal{R}$ is not an extension of $\mathcal{R}_0$ is immaterial. Assuming that $\mathcal{R}$ extends $\mathcal{R}_0$, we first abstract the part of $\mathcal{R}$ exceeding $\mathcal{R}_0$, so to obtain an abstract run $\mathcal{R}^\sharp$. This is done by abstracting every configuration in the run: times are abstracted with $\alpha_{\mathcal{T}_0}$, while untimed configurations are abstracted with α_X, where X is the set of the descendants of X_0 in the configuration at hand. The moves of $\mathcal{R}$ are mapped to abstract moves

in a natural way: moves not affecting the descendents of X_0, nor their relevant deposits or secrets, are not represented in the abstract run. Once the abstract run $\mathcal{R}^\sharp$ has been constructed, we apply $\Sigma_{\mathsf{A}}^{\#}(\mathcal{R}^\sharp)$ to obtain the next abstract actions. $\Sigma_{\mathsf{A}}(\mathcal{R})$ is defined as the concretisation of these actions. The concretisation of the adversary strategy $\Sigma_{\mathsf{Adv}}^{\#}$ can be defined in a similar way.

Theorem 1. *Starting from any abstract configuration, the relation $\to_\sharp$ is finitely branching, and it admits a finite number of runs.*

A direct consequence of Theorem 1 is that the abstract semantics is *finite-state*, and that *each abstract run is finite*. This makes the abstract LTS amenable to model checking.

Correspondence Between the Semantics. We now establish a correspondence between the abstract and the concrete semantics of BitML. Assume that we have a concrete run $\mathcal{R}_0$, representing the computation done so far. We want to observe the behaviour of a set of contracts X_0 in $\Gamma_{\mathcal{R}_0}$ (the last untimed configuration of $\mathcal{R}_0$). To this purpose, we run the abstract semantics, starting from an initial configuration $\Gamma_0^\sharp$, whose untimed component is $\alpha_{X_0}(\Gamma_{\mathcal{R}_0})$. The time component is obtained by abstracting the last time $\delta_{\mathcal{R}_0}$ in the concrete run. The parameter $\mathcal{T}_0$ used to abstract time is any finite superset of the deadlines occurring in contracts X_0 within $\Gamma_{\mathcal{R}_0}$. Hereafter we denote this set of deadlines as $\mathit{ticks}_{X_0}(\Gamma_{\mathcal{R}_0})$ (see Definition 8 in Appendix A).

When the contracts in X_0 evolve, the run $\mathcal{R}_0$ is extended to a run $\mathcal{R}$, which contains the descendents of X_0, i.e. those contracts whose *origin* belongs to X_0. These descendents are denoted with $\mathit{desc}_{\mathcal{R}_0}(\mathcal{R}, X_0)$.

Definition 5. *For all concrete runs $\mathcal{R}_0, \mathcal{R}$ such that $\mathcal{R}$ extends $\mathcal{R}_0$, and set of deposit names X_0, we define the set of deposit names $\mathit{desc}_{\mathcal{R}_0}(\mathcal{R}, X_0)$ as follows:*

$$\mathit{desc}_{\mathcal{R}_0}(\mathcal{R}, X_0) \;=\; \left\{ x \,\middle|\, \exists \Gamma', C, v : \Gamma_{\mathcal{R}} = \langle C, v\rangle_x \mid \Gamma' \text{ and } \mathit{orig}_{\mathcal{R}_0}(\mathcal{R}, x) \in X_0 \right\}$$

The following theorem states that the abstract semantics is a sound approximation of the concrete one. Every abstract run (conforming to A's abstract strategy $\Sigma_{\mathsf{A}}^{\#}$) has a corresponding concrete run (conforming to the concrete strategy derived from $\Sigma_{\mathsf{A}}^{\#}$). More precisely, each configuration $\Gamma^\sharp \mid T$ in the abstract run has a corresponding configuration in the concrete run, containing the concretization Γ of $\Gamma^\sharp$, besides a term Δ containing the parts unrelated to X_0. Further, each move in the abstract run corresponds to an analogous move in the concrete run.

Theorem 2 (Soundness). *Let $\mathcal{R}_0$ be a concrete run, let $X_0 \subseteq \mathrm{cn}(\Gamma_{\mathcal{R}_0})$, let $Z_0 \supseteq \mathcal{N}(X_0, \Gamma_{\mathcal{R}_0})$, let $\mathcal{T}_0 \in \wp_{fin}(\mathbb{N})$, let $\Gamma_0^\sharp = \alpha_{X_0,Z_0}(\Gamma_{\mathcal{R}_0}) \mid \alpha_{\mathcal{T}_0}(\Gamma_{\mathcal{R}_0})$. Let $\Sigma_{\mathsf{A}}^{\#}$ and $\Sigma_{\mathsf{Adv}}^{\#}$ be the abstract strategies of A and of Adv, and let $\Sigma_{\mathsf{A}} = \gamma(\Sigma_{\mathsf{A}}^{\#})$ and $\Sigma_{\mathsf{Adv}} = \gamma(\Sigma_{\mathsf{Adv}}^{\#})$ be the corresponding concrete strategies. For each abstract run $\Gamma_0^\sharp \to_\sharp^* \Gamma^\sharp \mid T$ conforming to $\Sigma_{\mathsf{A}}^{\#}$ and $\Sigma_{\mathsf{Adv}}^{\#}$, there exists a concrete run:*

$$\mathcal{R} \;=\; \mathcal{R}_0 \;\to^*\; \Gamma \mid \Delta \mid \min T$$

such that: (i) $\mathcal{R}$ *conforms to* Σ_{A} *and* Σ_{Adv}*; (ii)* Δ *contains all the subterms of* $\Gamma_{\mathcal{R}_0}$ *which are mapped to* 0 *when evaluating* $\alpha_{X_0,Z_0}(\Gamma_{\mathcal{R}_0})$*; (iii)* $\alpha_{X,Z_0}(\Gamma \mid \Delta) = \Gamma^\sharp$*, where* $X = desc_{\mathcal{R}_0}(\mathcal{R}, X_0)$*; (iv)* $\alpha_{\mathcal{T}_0}(\min T) = T$*; (v) the labels in* $\mathcal{R}$ *are the same as in* $\mathcal{R}^\sharp$*, except for the occurrences of* $y^\star$*.*

Note that soundness only guarantees the existence of some concrete runs, which are a strict subset of all the possible concrete runs. For instance, the concrete semantics also allows the non-observed part Δ to progress, and it contains configurations with a time $t \neq \min T$, for any T in any abstract run. Still, these concrete runs have an abstract counterpart, as established by the following completeness result (Theorem 3). This is almost dual to our soundness result (Theorem 2). Completeness maps concrete configurations to abstract ones using our abstraction functions for untimed configurations and time. Moreover, this run correspondence holds when the concrete strategy of A is derived from an abstract strategy, while no such restriction is required for the adversary strategy.

Theorem 3 (Completeness). *Let* $\mathcal{R}_0$ *be a concrete run, let* $X_0 \subseteq \mathrm{cn}(\Gamma_{\mathcal{R}_0})$*, let* $Z_0 \supseteq \mathcal{N}(X_0, \Gamma_{\mathcal{R}_0})$*, let* $\mathcal{T}_0 \supseteq ticks_{X_0}(\Gamma_{\mathcal{R}_0})$*, and let* $\Gamma_0^\sharp = \alpha_{X_0,Z_0}(\Gamma_{\mathcal{R}_0}) \mid \alpha_{\mathcal{T}_0}(\Gamma_{\mathcal{R}_0})$*. Let* $\Sigma_{\mathsf{A}}^\#$ *be the abstract strategy of* A*, and let* $\Sigma_{\mathsf{A}} = \gamma(\Sigma_{\mathsf{A}}^\#)$ *be the corresponding concrete strategy. For each concrete run* $\mathcal{R} = \mathcal{R}_0 \rightarrow^* \Gamma \mid t$ *conforming to* Σ_{A} *and to some* Σ_{Adv}*, there exists an abstract run:*

$$\mathcal{R}^\sharp \;=\; \Gamma_0^\sharp \;\rightarrow_\sharp^*\; \alpha_{X,Z_0}(\Gamma) \mid \alpha_{\mathcal{T}_0}(t)$$

such that: (i) $\mathcal{R}^\sharp$ *conforms to* $\Sigma_{\mathsf{A}}^\#$ *and to some* $\Sigma_{\mathsf{Adv}}^\#$*; (ii)* $X = desc_{\mathcal{R}_0}(\mathcal{R}, X_0)$*; (iii) if* $\mathcal{R} = \mathcal{R}_0 \rightarrow^* \Gamma' \mid t' \xrightarrow{\ell} \cdots$ *and* $\ell \in \Sigma_{\mathsf{A}}(\mathcal{R}_0 \rightarrow^* \Gamma' \mid t')$*, then there exists* ℓ' *such that* $\mathcal{R}^\sharp = \Gamma_0^\sharp \rightarrow_\sharp^* \Gamma^\sharp = \alpha_{X',Z_0}(\Gamma') \mid \alpha_{\mathcal{T}_0}(t') \xrightarrow{\ell'}_\sharp \cdots$ *where* $\ell' \in \Sigma_{\mathsf{A}}^\#(\Gamma_0^\sharp \rightarrow_\sharp^* \Gamma^\sharp)$ *and* $X' = desc_{\mathcal{R}_0}(\mathcal{R}_0 \rightarrow^* \Gamma' \mid t', X_0)$*.*

Example 5. Let $C = \texttt{reveal}\, a.\texttt{withdraw}\ \mathsf{A} + \texttt{put}\, y.\texttt{withdraw}\ \mathsf{B}$, and let $\mathcal{R}$ be the following concrete run, where the prefix $\cdots$ is immaterial (for simplicity, we also omit labels, times, and participants' strategies):

$$\begin{aligned}
\cdots &\rightarrow \langle C, 1\text{฿}\rangle_x \mid \langle \mathsf{B}, 1\text{฿}\rangle_y \mid \langle \mathsf{A}, 2\text{฿}\rangle_z \mid \{\mathsf{A} : a\#10\} \quad = \Gamma_0 \\
&\rightarrow \langle C, 1\text{฿}\rangle_x \mid \langle \mathsf{B}, 1\text{฿}\rangle_y \mid \langle \mathsf{A}, 2\text{฿}\rangle_z \mid \{\mathsf{A} : a\#10\} \mid \mathsf{B}[y \rhd \mathsf{B}] \\
&\rightarrow \langle C, 1\text{฿}\rangle_x \mid \langle \mathsf{B}, 1\text{฿}\rangle_y \mid \langle \mathsf{A}, 2\text{฿}\rangle_z \mid \{\mathsf{A} : a\#10\} \mid \mathsf{B}[y \rhd \mathsf{B}] \mid \mathsf{B}[y \rhd \mathsf{C}] \\
&\rightarrow \langle C, 1\text{฿}\rangle_x \mid \langle \mathsf{B}, 1\text{฿}\rangle_y \mid \langle \mathsf{A}, 2\text{฿}\rangle_z \mid \mathsf{A} : a\#10 \mid \mathsf{B}[y \rhd \mathsf{B}] \mid \mathsf{B}[y \rhd \mathsf{C}] \\
&\rightarrow \langle \texttt{withdraw}\ \mathsf{A}, 1\text{฿}\rangle_{x'} \mid \langle \mathsf{B}, 1\text{฿}\rangle_y \mid \langle \mathsf{A}, 2\text{฿}\rangle_z \mid \mathsf{A} : a\#10 \mid \mathsf{B}[y \rhd \mathsf{B}] \mid \mathsf{B}[y \rhd \mathsf{C}] \quad = \Gamma \\
&\rightarrow \langle \texttt{withdraw}\ \mathsf{A}, 1\text{฿}\rangle_{x'} \mid \langle \mathsf{C}, 1\text{฿}\rangle_{y'} \mid \langle \mathsf{A}, 2\text{฿}\rangle_z \mid \mathsf{A} : a\#10 \mid \mathsf{B}[y \rhd \mathsf{B}] \mid \mathsf{B}[y \rhd \mathsf{C}] \\
&\rightarrow \langle \mathsf{A}, 1\text{฿}\rangle_{x''} \mid \langle \mathsf{C}, 1\text{฿}\rangle_{y'} \mid \langle \mathsf{A}, 2\text{฿}\rangle_z \mid \mathsf{A} : a\#10 \mid \mathsf{B}[y \rhd \mathsf{B}] \mid \mathsf{B}[y \rhd \mathsf{C}]
\end{aligned}$$

By Theorem 3, this concrete run has the following corresponding abstract run w.r.t. $X_0 = \{x\}$. The initial configuration Γ_0 is abstracted w.r.t. X_0 and $Z_0 = \mathcal{N}(X_0, \Gamma_0) = \{a, y\}$. This causes deposit z to be neglected in the abstraction.

$$
\begin{aligned}
&\langle C, 1\ddot{\mathrm{B}}\rangle_x \mid \langle \mathsf{B}, 1\ddot{\mathrm{B}}\rangle_y \mid \{\mathsf{A} : a\#10\} \quad = \Gamma_0^\sharp \\
\to_\sharp\ &\langle C, 1\ddot{\mathrm{B}}\rangle_x \mid \langle \mathsf{B}, 1\ddot{\mathrm{B}}\rangle_y \mid \{\mathsf{A} : a\#10\} \mid \mathsf{B}[y, 0 \rhd y^\star] \\
\to_\sharp\ &\langle C, 1\ddot{\mathrm{B}}\rangle_x \mid \langle \mathsf{B}, 1\ddot{\mathrm{B}}\rangle_y \mid \mathsf{A} : a\#10 \mid \mathsf{B}[y, 0 \rhd y^\star] \\
\to_\sharp\ &\langle \texttt{withdraw}\ \mathsf{A}, 1\ddot{\mathrm{B}}\rangle_{x'} \mid \langle \mathsf{B}, 1\ddot{\mathrm{B}}\rangle_y \mid \mathsf{A} : a\#10 \mid \mathsf{B}[y, 0 \rhd y^\star] \quad = \Gamma^\sharp \\
\to_\sharp\ &\langle \texttt{withdraw}\ \mathsf{A}, 1\ddot{\mathrm{B}}\rangle_{x'} \mid \mathsf{A} : a\#10 \\
\to_\sharp\ &\mathsf{A} : a\#10
\end{aligned}
$$

We now compare the two runs. The concrete authorization for a self-donate of y is abstracted as an authorization for destroying y. Instead, the concrete authorization for donating y to C has no abstract counterpart. The concrete reveal of secret a and the subsequent contract move have identical abstract moves, which reach the abstract configuration $\Gamma^\sharp$. Technically, $\Gamma^\sharp$ is the result of abstracting the concrete configuration Γ w.r.t. $X' = \{x'\}$ and Z_0: here, we no longer abstract w.r.t. X_0, but instead use the set of its descendents X'. By contrast, the set Z_0 is unchanged. Note that, if we instead abstracted with respect to X_0, we would discard the contract x', in which case we could not perform the abstract step, because the abstract semantics does not discard x'. Similarly, if we instead used $Z' = \mathcal{N}(X', \Gamma) = \emptyset$ we would discard the secret a and the deposit y, invalidating the abstract steps. When Γ performs the next move (a donation) this is abstracted as a destroy move. Finally, the last concrete `withdraw` move is mapped to an abstract `withdraw` move, which does not create the deposit x''.

5 Verifying Liquidity

In this section we devise a verification technique for liquidity of BitML contracts, exploiting our abstract semantics. The first step is to give an abstract counterpart of liquidity: this is done in Definition 6, which mimics Definition 2, replacing concrete objects with abstract ones.

Definition 6 (Abstract liquidity). *Let* A *be an honest participant, with abstract strategy* $\Sigma^\#_\mathsf{A}$*, let* $\mathcal{R}_0^\sharp$ *be an abstract run, and let* X_0 *be a set of contract names in* $\Gamma_{\mathcal{R}_0^\sharp}$*. We say that* X_0 *is* $\sharp$-liquid w.r.t. $\Sigma^\#_\mathsf{A}$ in $\mathcal{R}_0^\sharp$ *if for all extensions* $\mathcal{R}^\sharp$ *of* $\mathcal{R}_0^\sharp$ *conforming to* $\Sigma^\#_\mathsf{A}$ *and to some* $\Sigma^\#_{\mathsf{Adv}}$*, there exists an extension* $\dot{\mathcal{R}}^\sharp = \mathcal{R}^\sharp \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_n}$ *of* $\mathcal{R}^\sharp$ *such that:*

$$\forall i \in 1..n : \ell_i \in \Sigma^\#_\mathsf{A}(\mathcal{R}^\sharp \xrightarrow{\ell_1}_\sharp \cdots \xrightarrow{\ell_{i-1}}_\sharp) \tag{3}$$

$$x \in \mathrm{cn}(\Gamma_{\dot{\mathcal{R}}^\sharp}) \implies \mathit{orig}_{\mathcal{R}_0^\sharp}(\dot{\mathcal{R}}^\sharp, x) \notin X_0 \tag{4}$$

To verify liquidity of a set of contracts X_0 in a concrete run $\mathcal{R}_0$, we will choose $\mathcal{R}_0^\sharp$ to be the run containing a single configuration $\Gamma_0^\sharp$, obtained by abstracting with α_{X_0} the last configuration of $\mathcal{R}_0$. In such case, the condition (4) above can be simplified by just requiring that $\mathrm{cn}(\Gamma_{\dot{\mathcal{R}}^\sharp}) = \emptyset$.

The following lemma states that abstract and concrete liquidity are equivalent. For this, it suffices that the abstraction is performed with respect to the contract names X_0, and to the set of deadlines occurring in the contracts X_0.

Lemma 2 (Abstract vs. concrete liquidity). *Let $\mathcal{R}_0$ be a concrete run, let $X_0 \subseteq \mathrm{cn}(\Gamma_{\mathcal{R}_0})$, and let $\mathcal{T}_0 = ticks_{X_0}(\Gamma_{\mathcal{R}_0})$. Let $\Gamma_0^\sharp = \alpha_{X_0}(\Gamma_{\mathcal{R}_0}) \mid \alpha_{\mathcal{T}_0}(\delta_{\mathcal{R}_0})$. Let $\Sigma_\mathsf{A}^\#$ be an abstract strategy (w.r.t. $\mathcal{T}_0$ and $\Gamma_0^\sharp$), and let $\Sigma_\mathsf{A} = \gamma_{\mathcal{R}_0}(\Sigma_\mathsf{A}^\#)$. Let $\mathcal{R}_0^\sharp = \Gamma_0^\sharp$ (i.e., the run with no moves). Then:*

$$X_0 \, is \, liquid \, w.r.t. \, \Sigma_\mathsf{A} \, in \, \mathcal{R}_0 \iff X_0 \, is \, \sharp\text{-}liquid \, w.r.t. \, \Sigma_\mathsf{A}^\# \, in \, \mathcal{R}_0^\sharp.$$

The following lemma states that if a contract is liquid w.r.t. some concrete strategy, then is also liquid w.r.t. some abstract strategy, and *vice versa*. Intuitively, this holds since if it is possible to make a contract evolve with a sequence of moves conforming to any concrete strategy, then the same moves can be also be generated by an abstract strategy.

Lemma 3. *Let $\mathcal{R}_0$ be a concrete run, and let $X_0 \subseteq \mathrm{cn}(\Gamma_{\mathcal{R}_0})$. X_0 is liquid w.r.t. some Σ_A in $\mathcal{R}_0$ iff X_0 is liquid w.r.t. $\gamma(\Sigma_\mathsf{A}^\#)$ in $\mathcal{R}_0$, for some $\Sigma_\mathsf{A}^\#$.*

Our main technical result follows. It states that liquidity is decidable, and that it is possible to automatically infer liquid strategies for a given contract.

Theorem 4 (Decidability of liquidity). *Liquidity is decidable. Furthermore, for any $\mathcal{R}_0$ and X_0, it is decidable whether there exists a strategy Σ_A such that X_0 is liquid w.r.t. Σ_A in $\mathcal{R}_0$. If such strategy exists, then it can be automatically inferred given $\mathcal{R}_0$ and X_0.*

Proof. Let A be an honest participant with strategy Σ_A, let $\mathcal{R}_0$ be a concrete run, and let X_0 be a set of contract names in $\Gamma_{\mathcal{R}_0}$. By Lemma 3, X_0 is liquid w.r.t. Σ_A iff there exists some abstract strategy $\Sigma_\mathsf{A}^\#$ such that X_0 is liquid w.r.t. $\Sigma'_\mathsf{A} = \gamma(\Sigma_\mathsf{A}^\#)$. By Lemma 2, X_0 is liquid w.r.t. Σ'_A iff X_0 is $\sharp$-liquid w.r.t. $\Sigma_\mathsf{A}^\#$. By Theorem 1, the abstract semantics is finite, and so the possible abstract strategies are finite. Therefore, $\sharp$-liquidity is decidable, and consequently also liquidity is decidable. Note that this procedure also finds a liquid strategy, if there exists one. □

6 Conclusions

We have developed a theory of liquidity for smart contracts, and a verification technique which is sound and complete for contracts expressed in BitML. Our finite-state abstraction can be applied, besides liquidity, to verify other properties of smart contracts. For instance, we could decide whether a strategy allows a participant to always terminate a contract within a certain deadline. Additionally, we could infer a strategy which guarantees that the contract terminates before a certain time (if any such strategy exists), or infer the strategy that terminates in the shortest time, etc. Although our theory is focussed on BitML, the

various notions of liquidity we have proposed could be applied to more expressive languages for smart contracts, like e.g. Solidity (the high-level language used by Ethereum). To the best of our knowledge, the only form of liquidity verified so far in Ethereum is the "strategyless multiparty" variant, which only requires the existence of a cooperative strategy to unfreeze funds (this property is analysed, e.g., by the Securify tool [35]). Since Ethereum contracts are Turing-powerful, verifying their liquidity is not possible in a sound and complete manner; instead, the reduced expressiveness of BitML makes liquidity decidable in that setting.

Acknowledgements. Massimo Bartoletti is partially supported by Aut. Reg. of Sardinia projects *Sardcoin* and *Smart collaborative engineering*. Roberto Zunino is partially supported by MIUR PON *Distributed Ledgers for Secure Open Communities*.

A Appendix

Lemma 4. *Let $\mathcal{R}_0, \mathcal{R}_1, \mathcal{R}_2$ be such that $\mathcal{R}_1$ extends $\mathcal{R}_0$ and $\mathcal{R}_2$ extends $\mathcal{R}_1$. Then:*

$$orig_{\mathcal{R}_0}(\mathcal{R}_1, orig_{\mathcal{R}_1}(\mathcal{R}_2, x)) \;=\; orig_{\mathcal{R}_0}(\mathcal{R}_2, x)$$

Proof of Lemma 4 (sketch). By induction on $\mathcal{R}_1$.

Definition 7 (Abstract semantics). *Let $\mathcal{T} \in \wp_{fin}(\mathbb{N})$. An* abstract configuration *is a term of the form $\Gamma \mid T$, where Γ is a concrete untimed configuration, and $T \in \operatorname{ran} \alpha_{\mathcal{T}}$. We then define the relation $\rightarrow_\sharp$ between abstract configurations by differences w.r.t. the concrete relation $\rightarrow$:*

1. *the rule* [C-Advertise] *is removed.*
2. *the rules for deposits are replaced by the following two rules:*

$$\frac{}{\langle \mathsf{A}, v\rangle_x \mid \Gamma \xrightarrow{\mathsf{A}:x,0,y^\star}_\sharp \langle \mathsf{A}, v\rangle_x \mid \mathsf{A}[x, 0 \rhd y^\star] \mid \Gamma}\ [\text{Dep-AbsAuthDestroy}]$$

$$\frac{}{\langle \mathsf{A}, v\rangle_x \mid \mathsf{A}[x, 0 \rhd y^\star] \mid \Gamma \xrightarrow{destroy(x,y^\star)}_\sharp \Gamma}\ [\text{Dep-AbsDestroy}]$$

3. *the rule* [Delay] *is replaced by the following:*

$$\frac{\delta = \min T' - \min T > 0}{\Gamma \mid T \xrightarrow{\delta}_\sharp \Gamma \mid T'}\ [\text{AbsDelay}]$$

4. *the rule* [C-Withdraw] *is replaced by the following:*

$$\frac{}{\langle \texttt{withdraw}\ \mathsf{A}, v\rangle_y \mid \Gamma \xrightarrow{withdraw(\mathsf{A},v,y)} 0}\ [\text{C-AbsWithdraw}]$$

5. the rule [Timeout] *is replaced by the following:*

$$\frac{D \equiv \texttt{after}\, t_1 : \cdots : \texttt{after}\, t_m : D' \qquad D' \not\equiv \texttt{after}\, t' : \cdots \qquad \langle D, v\rangle_x \mid \Gamma \xrightarrow{\ell}_\sharp \Gamma' \qquad x \in cv(\ell) \qquad \min T \geq t_1, \ldots, t_m}{\langle D + C, v\rangle_x \mid \Gamma \mid T \xrightarrow{\ell}_\sharp \Gamma' \mid T} \text{[AbsTimeout]}$$

Definition 8. *We define the function ticks from contracts to* $\wp_{fin}(\mathbb{N})$ *as follows:*

$$ticks(\textstyle\sum_{i\in I} D_i) = \bigcup_{i\in I} ticks(D_i) \qquad ticks(\mathsf{A} : D) = ticks(D)$$
$$ticks(\texttt{withdraw}\ \mathsf{A}) = \emptyset \qquad ticks(\texttt{after}\, t : D) = \{t\} \cup ticks(D)$$
$$ticks(\texttt{split}\ \boldsymbol{v} \to \boldsymbol{C}) = \bigcup ticks(\boldsymbol{C}) \qquad ticks(\texttt{put}\ \boldsymbol{x}\ \&\ \texttt{reveal}\ \boldsymbol{a}\ \texttt{if}\ p.\, C) = ticks(C)$$

Then, for any set of names X, we define the function $ticks_X$ *from concrete untimed configurations to* $\wp_{fin}(\mathbb{N})$ *as follows:*

$$ticks_X(\{G\}C) = \emptyset$$
$$ticks_X(\langle C, v\rangle_x) = \begin{cases} ticks(C) & if\, x \in X \\ \emptyset & otherwise \end{cases}$$
$$ticks_X(\langle \mathsf{A}, v\rangle) = ticks_X(\mathsf{A}[\chi]) = ticks_X(\{\mathsf{A} : a\#N\}) = ticks_X(\mathsf{A} : a\#N) = \emptyset$$
$$ticks_X(\Gamma \mid \Gamma') = ticks_X(\Gamma) \cup ticks_X(\Gamma')$$

Lemma 5. *If* $\mathcal{R} = \mathcal{R}_0 \to^* \Gamma \mid t$*, then* $ticks_{X_0}(\Gamma_{\mathcal{R}_0}) \supseteq ticks_{desc_{\mathcal{R}_0}(\mathcal{R}, X_0)}(\Gamma)$.

Proof of Lemma 5 (sketch). When a move is performed, a contract becomes syntactically smaller, hence the set of deposit names and secret names within the contract becomes a subset.

Definition 9 (Abstract strategies). *For any* $\mathcal{T} \in \wp_{fin}(\mathbb{N})$ *and initial abstract configuration* $\Gamma_0 \mid T_0$ *with* $T_0 \in \operatorname{ran} \alpha_{\mathcal{T}}$*, we define an* abstract strategy $\Sigma^{\#}_{\mathsf{A}}$ *as a PPTIME algorithm which takes as input an abstract run starting from* $\Gamma_0 \mid T_0$ *and a randomness source, and gives as output a finite sequence of actions. Abstract strategies are subject to same constraints imposed to concrete ones.*

Note that, since $\Sigma^{\#}_{\mathsf{A}}$ can only output moves according to the abstract semantics, it can only choose delays δ which jump from an interval T to a subsequent interval T', i.e. $\delta = \min T' - \min T$.

Proof of Theorem 1 (sketch). The theorem immediately follows from the definition of our abstract semantics, which, compared to the concrete semantics, removes or abstracts all the BitML rules which can violate the statement. More precisely, using rule induction we observe that each abstract step makes the configuration syntactically "smaller", ensuring termination. Further, we have a finite amount of rules, and each rule can only cause a finite amount of branches.

Proof of Theorem 2 (sketch). Essentially, the concrete run can perform the same moves of the abstract run, with the following minor changes. The abstract

rules for destroying deposits (and the related authorizations) involve the name $y^\star$, which are replaced by fresh names y in the concrete run. Further, abstract delay moves change the abstract time T to T': in the concrete run, instead, we make time move from $\min T$ to $\min T'$. This makes the concrete and abstract timeout rules to agree on which branches $\texttt{after}\, t : D$ are enabled.

Proof of Theorem 3 (sketch). Each concrete move corresponds to zero or more abstract moves: in the latter case, the concrete and abstract moves are related as follows: (i) contract moves are unchanged; (ii) all authorizations are unchanged, but for $\mathsf{A} : x, \mathsf{B}$ (generated by [Dep-AuthDonate]) which is abstracted as $\mathsf{A} : x, 0, y^\star$; (iii) deposit moves affecting a set Y of deposits are transformed to a sequence of [Dep-AbsDestroy] moves, destroying those deposits in Y which are present in the abstract configuration; (iv) reveal moves are unchanged; (v) delay moves are mapped to delay moves (not necessarily of the same duration).

Proof of Lemma 2. See [15].

Proof of Lemma 3 (sketch). The lemma holds since $\Sigma^\#_\mathsf{A}$ can be defined in terms of Σ_A, in such a way to preserve the following invariant: each conforming run to $\Sigma^\#_\mathsf{A}$ can be transformed into a concrete run conforming to Σ_A. Upon receiving a (conforming) abstract run, if some descendent of X_0 is still present, $\Sigma^\#_\mathsf{A}$ computes a corresponding concrete run and queries $\Sigma^\#_\mathsf{A}$ with it, learning the next concrete moves. Since X_0 is liquid, the concrete strategy eventually must perform a move which is relevant for the contracts X_0, and that move can then be chosen by $\Sigma^\#_\mathsf{A}$. If such move is then taken by the abstract adversary, the invariant is clearly preserved. If instead the adversary takes another move, we can extend the concrete run accordingly, and still preserve the invariant.

Liquidity for Finite LTS. We now give an alternative characterization of liquidity, which corresponds to Definition 2 on transition systems with finite traces, like the one obtained through the abstraction introduced in Sect. 4.

Definition 10 (Maximal run). *We say that a run $\mathcal{R}$ is maximal w.r.t. a set of strategies $\boldsymbol{\Sigma}$ when $\mathcal{R} \xrightarrow{\ell}$ implies $\ell \notin \boldsymbol{\Sigma}(\mathcal{R})$.*

Definition 11 (Liquidity for finite LTS). *Assume that A is the only honest participant, with strategy $\Sigma^\#_\mathsf{A}$. We say that X_0 is $\sharp_{fin}$-*liquid w.r.t. $\Sigma^\#_\mathsf{A}$ in $\mathcal{R}^\sharp_0$ *when, for all extensions $\mathcal{R}^\sharp$ of $\mathcal{R}^\sharp_0$ conforming to $\Sigma^\#_\mathsf{A}$ (and to some $\Sigma^\#_{\mathsf{Adv}}$), if $\mathcal{R}^\sharp$ is maximal w.r.t. $\Sigma_\mathsf{A}, \Sigma_{\mathsf{Adv}}$ and $x \in \mathrm{cn}(\Gamma_{\mathcal{R}^\sharp})$, then $orig_{\mathcal{R}^\sharp_0}(\mathcal{R}^\sharp, x) \notin X_0$.*

Lemma 6. *X_0 is $\sharp$-liquid w.r.t. $\Sigma^\#_\mathsf{A}$ in $\mathcal{R}^\sharp_0$ iff X_0 is $\sharp_{fin}$-liquid w.r.t. $\Sigma^\#_\mathsf{A}$ in $\mathcal{R}^\sharp_0$.*

Proof. For the "only if part", assume that X_0 is $\sharp$-liquid w.r.t. $\Sigma^\#_\mathsf{A}$ in $\mathcal{R}^\sharp_0$, and let $\mathcal{R}^\sharp$ be a maximal extension (w.r.t. $\Sigma^\#_\mathsf{A}, \Sigma^\#_{\mathsf{Adv}}$) of $\mathcal{R}^\sharp_0$ conforming to $\Sigma^\#_\mathsf{A}, \Sigma^\#_{\mathsf{Adv}}$. By Definition 6, condition (3) can only hold for $\dot{\mathcal{R}}^\sharp = \mathcal{R}^\sharp$. Hence, for all $x \in \mathrm{cn}(\Gamma_{\mathcal{R}^\sharp})$, by condition (4) it follows that $orig_{\mathcal{R}^\sharp_0}(\mathcal{R}^\sharp, x) \notin X_0$.

For the "if part", assume that X_0 is $\sharp_{fin}$-liquid w.r.t. $\Sigma^{\#}_{\mathsf{A}}$ in $\mathcal{R}^{\sharp}_0$, and let $\mathcal{R}^{\sharp}$ be an extension of $\mathcal{R}^{\sharp}_0$ conforming to $\Sigma^{\#}_{\mathsf{A}}, \Sigma^{\#}_{\mathsf{Adv}}$. There are two cases:

- If $\mathcal{R}^{\sharp}$ is maximal w.r.t. $\Sigma^{\#}_{\mathsf{A}}, \Sigma^{\#}_{\mathsf{Adv}}$, then by Definition 11 it follows that $x \in \mathrm{cn}(\Gamma_{\mathcal{R}^{\sharp}})$ implies $orig_{\mathcal{R}^{\sharp}_0}(\mathcal{R}^{\sharp}, x) \notin X_0$. Hence, conditions (3)–(4) of Definition 6 follow by choosing $\dot{\mathcal{R}}^{\sharp} = \mathcal{R}^{\sharp}$.
- If $\mathcal{R}^{\sharp}$ is *not* maximal w.r.t. $\Sigma^{\#}_{\mathsf{A}}, \Sigma^{\#}_{\mathsf{Adv}}$, let $\dot{\mathcal{R}}^{\sharp}$ be the longest extension of $\mathcal{R}^{\sharp}$ made only by moves conforming to $\Sigma^{\#}_{\mathsf{A}}$. Let $\dot{\Sigma}^{\#}_{\mathsf{Adv}}$ be the strategy which (i) is equal to $\Sigma^{\#}_{\mathsf{Adv}}$ on the prefix $\mathcal{R}^{\sharp}$, (ii) permits A's action on the extension, (iii) forbids any action after $\dot{\mathcal{R}}^{\sharp}$. By this construction, $\dot{\mathcal{R}}^{\sharp}$ is maximal w.r.t. $\Sigma^{\#}_{\mathsf{A}}, \dot{\Sigma}^{\#}_{\mathsf{Adv}}$. So, by Definition 11 we have $orig_{\mathcal{R}^{\sharp}_0}(\dot{\mathcal{R}}^{\sharp}, x) \notin X_0$ for all $x \in \mathrm{cn}(\Gamma_{\dot{\mathcal{R}}^{\sharp}})$. Conditions (3)–(4) of Definition 6 follow by choosing $\dot{\mathcal{R}}^{\sharp}$. □

References

1. Understanding the DAO attack, June 2016. http://www.coindesk.com/understanding-dao-hack-journalists/
2. Parity Wallet security alert, July 2017. https://paritytech.io/blog/security-alert.html
3. A Postmortem on the Parity Multi-Sig library self-destruct, November 2017. https://goo.gl/Kw3gXi
4. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, Ł.: Fair two-party computations via Bitcoin deposits. In: Böhme, R., Brenner, M., Moore, T., Smith, M. (eds.) FC 2014. LNCS, vol. 8438, pp. 105–121. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44774-1_8
5. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, Ł.: Modeling Bitcoin contracts by timed automata. In: Legay, A., Bozga, M. (eds.) FORMATS 2014. LNCS, vol. 8711, pp. 7–22. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10512-3_2
6. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. In: IEEE S & P, pp. 443–458 (2014). First appeared on Cryptology ePrint Archive. http://eprint.iacr.org/2013/784
7. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. Commun. ACM **59**(4), 76–84 (2016)
8. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum Smart Contracts (SoK). In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 164–186. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
9. Atzei, N., Bartoletti, M., Cimoli, T., Lande, S., Zunino, R.: SoK: unraveling bitcoin smart contracts. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 217–242. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_9
10. Atzei, N., Bartoletti, M., Lande, S., Zunino, R.: A formal model of Bitcoin transactions. In: Meiklejohn, S., Sako, K. (eds.) FC 2018. LNCS, vol. 10957, pp. 541–560. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-58387-6_29

11. Banasik, W., Dziembowski, S., Malinowski, D.: Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9879, pp. 261–280. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45741-3_14
12. Bartoletti, M., Cimoli, T., Zunino, R.: Fun with Bitcoin smart contracts. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 432–449. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_32
13. Bartoletti, M., Zunino, R.: Constant-deposit multiparty lotteries on Bitcoin. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 231–247. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_15
14. Bartoletti, M., Zunino, R.: BitML: a calculus for Bitcoin smart contracts. In: ACM SIGSAC CCS, pp. 83–100. ACM (2018)
15. Bartoletti, M., Zunino, R.: Verifying liquidity of Bitcoin contracts. Cryptology ePrint Archive, Report 2018/1125 (2018). https://eprint.iacr.org/2018/1125
16. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_7. http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf
17. Bentov, I., Kumaresan, R.: How to use Bitcoin to design fair protocols. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 421–439. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44381-1_24
18. Bhargavan, K., et al.: Formal verification of smart contracts. In: PLAS (2016)
19. Buterin, V.: Ethereum: a next generation smart contract and decentralized application platform (2013). https://github.com/ethereum/wiki/wiki/White-Paper
20. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: scaling byzantine agreements for cryptocurrencies. In: Symposium on Operating Systems Principles, pp. 51–68 (2017)
21. Grishchenko, I., Maffei, M., Schneidewind, C.: Foundations and tools for the static analysis of Ethereum smart contracts. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 51–78. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_4
22. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) POST 2018. LNCS, vol. 10804, pp. 243–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_10
23. Hildenbrandt, E., et al.: KEVM: a complete formal semantics of the Ethereum Virtual Machine. In: IEEE Computer Security Foundations Symposium (CSF), pp. 204–217. IEEE Computer Society (2018)
24. Hirai, Y.: Defining the Ethereum Virtual Machine for interactive theorem provers. In: Brenner, M., et al. (eds.) FC 2017. LNCS, vol. 10323, pp. 520–535. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_33
25. Klomp, R., Bracciali, A.: On symbolic verification of Bitcoin's script language. In: Garcia-Alfaro, J., Herrera-Joancomartí, J., Livraga, G., Rios, R. (eds.) DPM/CBT -2018. LNCS, vol. 11025, pp. 38–56. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00305-0_3
26. Kumaresan, R., Bentov, I.: How to use Bitcoin to incentivize correct computations. In: ACM CCS, pp. 30–41 (2014)
27. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: ACM CCS, pp. 254–269 (2016)

28. Maxwell, G.: The first successful zero-knowledge contingent payment (2016). https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/
29. Miller, A., Bentov, I.: Zero-collateral lotteries in Bitcoin and Ethereum. In: EuroS&P Workshops, pp. 4–13 (2017)
30. Miller, A., Cai, Z., Jha, S.: Smart contracts and opportunities for formal methods. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 280–299. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_22
31. Mythril (2018). https://github.com/ConsenSys/mythril
32. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008). https://bitcoin.org/bitcoin.pdf
33. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-orderlogic, vol. 2283. Springer Science & Business Media, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
34. Rocket, T.: Snowflake to avalanche: a novel metastable consensus protocol family for cryptocurrencies (2018). https://avalanchelabs.org/avalanche.pdf
35. Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: practical security analysis of smart contracts. In: ACM CCS, pp. 67–82 (2018)