Bitcoin Trace-Net:
Formal Contract Verification at Signing Time.

James Chiang
james.chiang@protonmail.com

Abstract

Contract protocols which enable the scaling of the Bitcoin network or increase the privacy of on-chain transactions are constructed with multi-party contract transactions, which must be negotiated, signed and broadcast by multiple participants according to protocol specification. We propose Bitcoin Trace-Net as an automated and generalized formal verification method for all contracting protocol types. Verification without any specification of the underlying protocol can be performed against a generalized contract correctness policy at signing time. This generalization of signing safety enables key management interoperability between different contracting protocols, which in turn has the potential to improve both scalability and privacy on the Bitcoin network.

1 Introduction

Although the expressiveness of the Bitcoin scripting language is comparatively limited, it has nonetheless been successfully used to construct multi-party contracting protocols which increase both scalability and privacy of transactions on the Bitcoin network. Well-known examples include the Lightning Network [1] and Coinjoin [2] protocol variants. However, because the protocol designs differ significantly and are designed in an ad hoc manner, both contract verification and generalized key management methods have been difficult to generalize across protocols thus far. Furthermore, there have been cases where the protocol specification itself has been insufficiently verified and has led to vulnerabilities in subsequent implementations, as was the case with the Lightning network [3].

Generalized contract verification at run-time is desirable because it provides formal contract verification across all protocol types. It also allows key management to be decoupled from the specifics of any protocol logic. A generalized key management implementation can thereby check contract transactions at signing time and ensure these correctly adhere to a generalized signing policy, as depicted in figure 1.

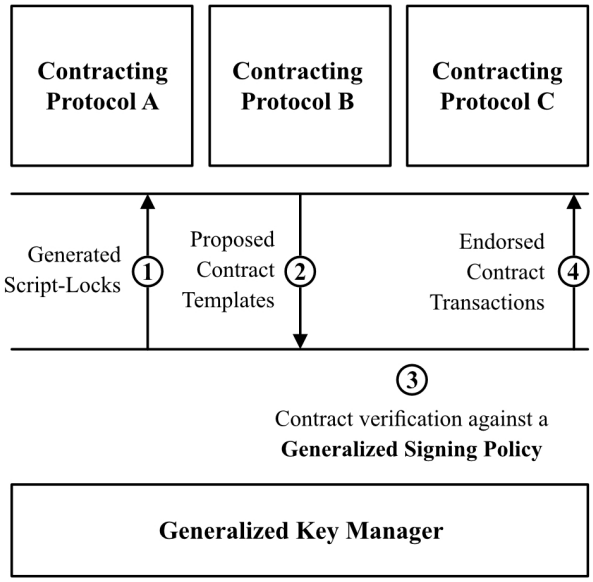

Figure 1: Trace-Net enables automated contract verification at signing time against a generalized signer policy, allowing key management to be generalized across different contracting protocols.

Multi-party contract protocols are currently supported with dedicated key management implementations. However, in order to share liquidity between protocol clients, funds must either be transferred on-chain between wallets or secret keys must be shared between applications: Both options are undesirable because they result in unnecessary on-chain transactions or risk the potential exposure of private keys.

In this paper, we propose a formalism named Bitcoin Trace-Net, which enables the generalized evaluation of contract safety of any contracting protocol at run-time. Bitcoin Trace-Net models are constructed from a set of proposed Bitcoin

transactions before signing, as shown in step 2 of figure 1. Such a model describes the possible contract state-space in its entirety, allowing for analysis of all feasible contract execution paths. Trace-Net contract execution traces express the properties we wish to verify to ensure a contract is safe according to a predefined signing policy.

Towards this goal we accept a necessary compromise in the expressiveness of Bitcoin script, by limiting our usage to a Bitcoin script subset encoded by a template language for which static analysis can be automated. Miniscript [4] [5] [6] [7] is a recently proposed template language for Bitcoin script and we restrict Trace-Net verification to contracts which feature Miniscript compatible output scripts. Such a statically analyzable template language is necessary for automated contract verification, because automating the determination of all satisfaction paths for every possible Bitcoin script is not known to be feasible.

1.1 Related Work

Contract verification methods of smart contracts which can be executed on the Ethereum EVM have been demonstrated with symbolic execution [8], theorem proving [9] [10] and model-checking [11] [12] approaches . Given that contracts in Ethereum are persistent objects which exhibit potentially unbounded state-spaces, traditional software verification approaches can often be successfully applied. There have also been efforts to specify Ethereum smart contracts as finite-state automata [13], thereby formalising contract behavior and safety.

Bitcoin contracts, however, are frequently constructed with multiple script instances across different transactions, where state computed from one script evaluation cannot be directly carried over to subsequent child transactions. Furthermore, the nature of Bitcoin script makes static-analysis difficult. Alternative scripting languages developed for Bitcoin-like protocols such as Simplicity [14] can be parsed at run-time, but are generally incompatible with Bitcoin script today. However, the recent introduction of Miniscript has made the automated determination of script satisfaction paths possible, which crucially enables the full contract state-space modeling proposed by Trace-Net.

Trace-Net also adopts classical Petri Net language [15] elements to model the execution of on-chain transactions. The Petri Net formalism has previously been applied to derive Bitcoin address ownership clusters [16], but not for modeling Bitcoin contract states. Trace-Net models the availability of unspent outputs in Bitcoin contracts with classical Petri Net language elements, but extends this skeleton with additional state and state transition rules to represent the full Bitcoin contract state-space.

However, unlike related work which aims to formally verify smart contract designs, Trace-Net is designed to enable automated verification of Bitcoin contracts at signing time against a protocol-agnostic signing policy.

1.2 Introduction to Bitcoin Contracts

In this section we give a short overview of the construction, update and execution of Bitcoin contracts, which are enabled by Bitcoin script-locks, parent-child transaction structures and transaction time-locks.

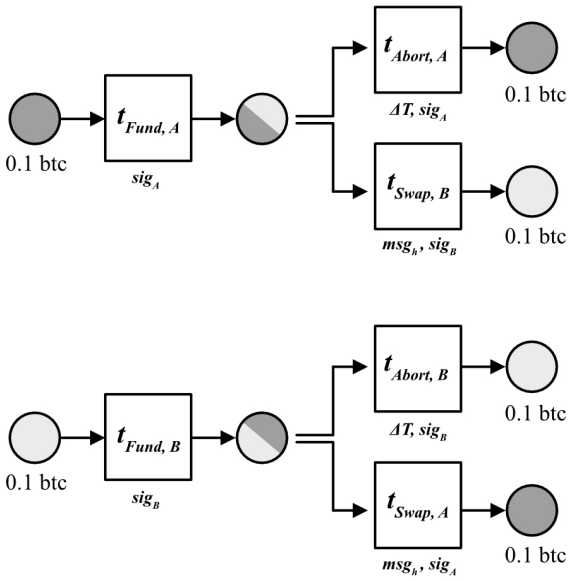

Figure 2: An atomic swap contract allows two parties (A, B) to swap ownership of coins. Both parties confirm funding transactions which send funds to an output with two satisfaction paths: A time-out and a swap path. The swap paths of both funding transaction outputs feature the same hash-lock, thereby atomically linking the two paths. The hash-lock is generated by the initiating party (A), who confirms the funding transaction first, and only reveals the message to the counter-party (B) when executing the swap transaction $t_{swap,A}$, thereby enabling $t_{swap,B}$

The scripting language of Bitcoin is a forth-like language which manipulates the upper regions of primary and alternative script interpreter stacks during evaluation. Bitcoin script is non-Turing complete but expressive enough to allow for basic computation and the construction of simple cryptographic primitives. Table 1 shows a non-exhaustive list of possible output script-locks expressible by a Bitcoin script instance. An output script-lock is unlocked by its satisfaction when the output is spent by a child transaction. Bitcoin script also features non-looping flow-control, allowing for logical combinations of script-locks and alternative script satisfaction paths.

Each Bitcoin output is currently encumbered with a single[1] output script.

Script-Lock Type	Satisfaction
Public Key	Signature
Hash Digest	Hash Preimage
After Timelock	Minimum global time passed
Older Timelock	Minimum age of output passed

Table 1: Examples of output locks expressible in script.

Unlike a contract object in the Ethereum EVM, however, a Bitcoin output script is only executed once when the output is spent by a child transaction. It is not possible for a single Bitcoin output script instance to experience more than one state transition: It is either unspent or spent, the latter requiring a child transaction with valid script inputs which satisfy the specific sequence of script-locks. The child transaction provides these script inputs in its input witness fields.

Instead, multi-party contracting protocols with larger state space designs must be constructed with multiple parent-child transactions or with transactions atomically linked by cryptographic primitives. Such parent-child designs generally involve contract output scripts with multiple possible satisfaction paths, each spendable by different input arguments.

This is shown in figure 2, which depicts an atomic swap contract between two parties. We chose this contract type as an example throughout this paper for its simplicity and representativeness. An atomic swap enables two parties to swap coins and features two funding contract transaction templates, each featuring output scripts with two satisfaction paths: An abort and a successful swap path. Both swap paths of the two outputs feature the same hash-lock, which is generated by the initiating party hashing a secret message. It also is the initiating party who must first confirm its funding transaction. Once both funding transactions are confirmed, the broadcast of the swap transaction by the initiating party will necessarily reveal the hash message on-chain, thereby enabling the counter-party to execute its swap transaction as well. If the initiating party never executes its swap transaction and decides to wait and confirm the abort transaction, so can the counter-party.

This example illustrates how a Bitcoin contract can be constructed and updated through-out its life-cycle, thereby representing a larger contract state than a single Bitcoin script instance could achieve. We note that an atomic swap with hash-locks provides only limited privacy gains, as the two transactions can be correlated on-chain with its identical hash-locks. Nonetheless, hash-locks are used to atomically couple satisfaction paths in more complex protocols such as in the Lightning Network. With the upcoming introduction of the Schnorr [19] signature scheme with Bitcoin Taproot, hash-locks can be replaced with adaptor signatures, which allow for the atomic coupling of two different public key, adaptor signature pairs. Since the adaptor signatures are never revealed on-chain, such an atomic swap would be entirely invisible to the on-chain observer. This script-lock type can easily be adopted by Trace-Net with identical firing semantics as hash-locks.

[1]The potential introduction of Bitcoin Taproot [17] [18] will enable multiple alternative script paths for each output, but the logical equivalent can still be expressed by a single script.

1.3 Introduction to Miniscript

Having provided an illustrative example of a multi-party contract design in Bitcoin, we proceed to introduce a Bitcoin script template language for which we can automate script analysis to determine both output script safety and valid satisfaction paths.

Miniscript is a template language which encodes a subset of Bitcoin script and facilitates its analysis at run-time. As a template language it is fully compatible with Bitcoin consensus, but sacrifices a degree of expressiveness compared to the full Bitcoin script language. Most importantly, however, it is possible to determine the different satisfaction paths for a given Miniscript encoded output script and what input argument types are required for spending. These arguments are provided by the witness data in the input of the spending transaction, and must evaluate to a valid script interpreter stack state to be valid. In Bitcoin, a consensus-valid script interpreter stack state is one with a non-zero top element.

In effect, the Miniscript template language allows us to parse for the information necessary to determine each unique satisfaction path and reconstruct its valid input argument sequence, even if the script-locks in each satisfaction path were not generated by the verifier. Each satisfaction path must have at least one public key script-lock and may also include any combination of Miniscript script-locks shown in table 2.

Miniscript Script-Lock Type	Satisfaction
Public Key	Signature
Public Key Hash	Public Key + Signature
RIPEMD160 Digest	RIPEMD160 Message
HASH160 Digest	HASH160 Message
SHA256 Digest	SHA256 Message
HASH256 Digest	HASH256 Message
After Timelock	Minimum global time
Older Timelock	Minimum age of output
Required Input Data	Byte literal

Table 2: Miniscript script-lock types.

In addition to script-lock types, Miniscript also features

boolean expressions which allow for the construction of alternate satisfaction paths. This means that an output script can encode multiple, alternate combinations of script-locks, as shown in figure 3. We capture this notion in the following definition: For a contract transaction output O_j at index j, we can define a function *SatPaths* which returns a witness encoding w_k for each possible satisfaction path of the Miniscript-encoded output O_j.

- $SatPaths(O_j) := \{w_0, w_1, ..., w_n\}$
- $w_k := [(Lock_0, Type_0), (Lock_0, Type_0), ..., (Lock_k, Type_k)]$

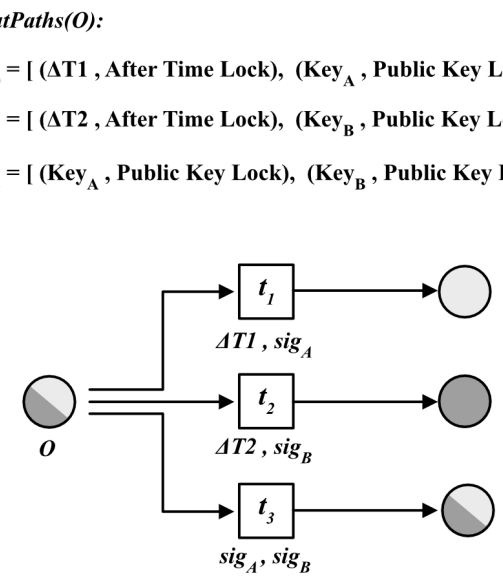

Figure 3: An output O with a Miniscript compatible output script can be statically analyzed for all its satisfaction paths, denoted $SatPaths(O)$. These satisfaction paths can be encoded as witness encodings w and imply possible spending transaction types.

Each witness encoding represents a unique output satisfaction path in form of a sequence of script-locks and their type. A contract participant who can satisfy every individual script-lock of w_k can therefore unlock w_k, since the participant can independently produce a valid, satisfying witness corresponding to the satisfaction path of w_k. The sequence of w_k indicates the ordering of witness data elements required for script satisfaction.

We note that the satisfaction of time-locks in a given witness encoding w_k is not specific to any contract participant and does not require the production of any satisfying transaction witness data, as it simply implies a temporal constraint on the confirmation of the transaction.

2 Bitcoin Trace-Net Overview

The Trace-Net formalism enables the automated analysis of a set of unsigned transaction templates to be formally safe for the contract verifier, before either participant endorses the contract or contract update. Contract safety can initially be defined as compliance with a generalized policy which denotes an expected balance unilaterally with-drawable by the verifying actor. This contract execution path must also be robust against any possible withholding of data or broadcasting of competing transactions by the counter-party. Section 3.1 defines generalized policy dimensions enabled by the definition of contract execution properties verifiable with Trace-Net.

Transaction templates are built from script-locks generated by both actors negotiating the contract (See step 1 in figure 1). Before providing signatures and hash messages (preimages) to the counter-party, each actor will formally verify which potential contract execution paths are safe for the given transaction templates.

$$\sigma = z_0 \xrightarrow{\theta_0} z_1 ... z_{n-1} \xrightarrow{\theta_{n-1}} z_n$$

The goal of a Trace-Net evaluation is the determination whether an execution trace σ exists which begins at an initial contract state z_0 and eventually leads to a safe final state z_n. In the following subsections of section 2, we introduce the firing semantics which determine which contract state transitions are fire-able at each contract state z and from which actor they can be initiated by. This is necessary in order to generate the entire contract state space (section 2.6), which is then examined for the presence of safe contract execution traces, as detailed in section 3.

2.1 Internal and External Actors

An actor in the Trace-Net model is a participant of a multi-party contracting protocol. An actor generates public keys, public key hashes, hash-locks, shared public keys and other cryptographic primitives which function as output script-locks with which the contract transaction templates are then constructed.

Each actors participating in a contract protocol generates and controls a subset of the secrets which can satisfy or unlock script-locks featured in the negotiated contract transaction templates and involved outputs. During the life-cycle of a multi-party contract, transaction templates may be added and secrets to individual output script-locks shared between actors.

In Trace-Net, we only distinguish between the *internal* and *external* actors. There may be more participants in a contracting protocol, but the verifying actor can only definitively distinguish whether a script-lock was locally generated and whether the script-lock satisfaction was shared with an external actor. In other words, the verifier must assume all

script-locks which it does not exclusively control are controlled by a single external actor. Note, that the ability of an actor to unlock a script-lock may change during the contract life-cycle, as signatures are broadcast and hash messages (preimages) are shared between actors.

Importantly, Trace-Net formalizes contract states in which actors can produce a valid witness for a given contract transaction throughout the contract life-cycle. We can define $CanSat(w, actor)$ where $actor \in \{int, ext\}$ and w is a witness encoding as defined in section 1.3:

$$CanSat(w, actor) := \begin{cases} 1 & iff\ actor\ can\ satisfy\ w \\ 0 & Otherwise \end{cases}$$

We explicitly note that time-locks which are part of a witness w encoding are not considered in the evaluation of $CanSat(w, actor)$ as they are not actor-specific.

The value of $CanSat(w, actor)$ for a given w and $actor$ can change between contract states. The sharing of contract relevant signatures and preimages between actors are considered a contract transition type and defined in the following section.

2.2 Contract Transitions

We define three types of transitions in a Bitcoin contract which imply a change in contract state when they are fired.

- **e** - Off-chain Contract Transitions
- **t** - On-chain Contract Transitions
- **d** - Time Transitions

All transition types are atomic. **e** transition types are fire-able anytime by the actor which generated the script-lock, and represents the sharing of the script-lock satisfaction with the counter-party. **t** transition types are fire-able by the actor which can produce satisfying witnesses for all inputs of the represented transaction. **d** type time transitions represent the time delays which must pass to release time script-locks. The remainder of this section details the firing rules of off-chain transitions. Firing rules for on-chain and time transitions are defined in subsequent sections 2.3 and 2.6.

An off-chain event transition occurs, when an actor directly or indirectly[2] reveals information to the other, thereby giving the receiving actor the ability to unlock an output script lock. Each off-chain transition is related to a specific output script-lock in a contract and can only be fired once. Let us consider a public key A in an output script generated by the internal actor. This script-lock can be satisfied by the internal actor anytime, as it controls the secret key a. If the internal actor shares the corresponding signature $sig_A(t)$ for the contract transaction t with the external actor, the external actor can now also produce the satisfying witness element for that transaction. This change in the external actor's ability to unlock this specific script-lock implies a change in the contract state, and is denoted with $e^{int}_{sig_A(t)}$. The superscript denotes the internal actor who can fire this off-chain event transition. If a signature is broadcast on-chain as part of a valid transaction, the corresponding off-chain transition can no longer fire, as the signature has been implicitly shared. Finally, each off-chain contract transition can only fire one time, as previously secret information can only be revealed once.

We can also consider a hash-lock digest in an output script generated by the internal or external actor. If the hash-lock h is generated by the external actor and its message shared with the internal actor, the contract transition $e^{ext}_{msg_h}$ is fired. Unlike a signature, a hash-lock is not bound to a specific transaction, and can be featured in more than one output script.

Finally, we illustrate an off-chain contract transition for a shared public key P which is created by adding public key points P_{int} and P_{ext}, each generated by the internal and external actors respectively. P is featured as a output script-lock in transaction t. Initially, neither actor has the full secret private key. However, if an actor shares its half of the private key secret, the other actor will be able to reconstruct the full secret and obtain the ability to produce $sig_P(t)$. An actor's revealing of its half of the shared public key P is represented by the contract state transitions $e^{int}_{key_P}$ and $e^{ext}_{key_P}$ respectively.

We can now summarize the types of off-chain transition types in the following table 3. We reiterate that each feasible off-chain event transition for each unique output script-lock can only fire once in the Trace-net model. Simultaneously, an off-chain contract transition is fire-able in any contract state where it has never been fired before. We also note that additional script-lock types can be modeled with Trace-Net but are omitted here, such as public key pairs which are atomically linked with adaptor signatures enabled by the Schnorr signature scheme.

Output Script-Lock	Feasible Off-chain Transitions
Int. Public Key P in tx t	$e^{int}_{sig_P(t)}$
Ext. Public Key P in tx t	$e^{ext}_{sig_P(t)}$
Shared Public Key P	$e^{int}_{key_P}$, $e^{ext}_{key_P}$
Int. Hash-lock h	$e^{int}_{msg_h}$
Ext. Hash-lock h	$e^{ext}_{msg_h}$

Table 3: Off-chain Contract Transition Types.

[2]The message/preimage to a hash-lock may be revealed on-chain when an output is spent, therby implying the firing of an off-chain transition in a contract featuring the same hash-lock.

2.3 On-chain Contract Transitions

In this section we detail the Trace-Net firing rules for on-chain contract transitions. Whilst off-chain and time transitions provide determinisms which describe contract states in which output script-locks can be satisfied by actors, we must also introduce a model which describes the availability of unspent outputs on the Bitcoin block-chain. To this end, Trace-Net adopts a classic Petri Net skeleton to capture the state and fire-ability of on-chain transitionsgit based on the availability of unspent outputs.

2.3.1 Petri Net Output State Model

There are four classical Petri-Net elements [15] adopted by the Trace-Net skeleton model: Places, tokens, arcs and on-chain transitions, as shown in figure 4. Tokens can only exist in places. Each on-chain transition is connected to at least one place with directed arcs. Arcs pointing from a place to an on-chain transition reflect transaction inputs and are denoted input arcs $arc(p,t)$. Arcs pointing to places represent transaction outptus and are denoted output arcs $arc(t,p)$. A given place p can only connect to a single input and single output arc at most and represents a transaction output. The Trace-Net skeleton (T,F,P) represents the sets of Petri Net transitions, arcs and places respectively for a given Bitcoin contract under evaluation.

When an on-chain transition fires, the input arcs of the fired transition each consume a token from their connected place. Simultaneously, each transition output arc will produce a token to its connected place during firing. Each place representing a transaction output *Out* adopts the output amount and witness encodings $SatPaths(Out)$.

The availability of confirmed and unspent outputs are modelled by tokens. The place marking m is defined as the number of tokens currently present in the set of places P.

$$m(p) = \{0,1\}, where\ p \in P$$

It is a necessary condition for the firing of an on-chain transition that connected input arcs consume from places populated by tokens. A Bitcoin transaction can only be confirmed if it is spending unspent transaction outputs: A resource modelled by availability of tokens in places connected to by a transitions input arcs. t^- denotes the required token availability for t to fire:

$$t^- = \begin{cases} 1 & if\ arc(p,t) \in F \\ 0 & if\ arc(p,t) \notin F \end{cases}$$

We denote the token availability requirement for the firing of on-chain transitions as follows:

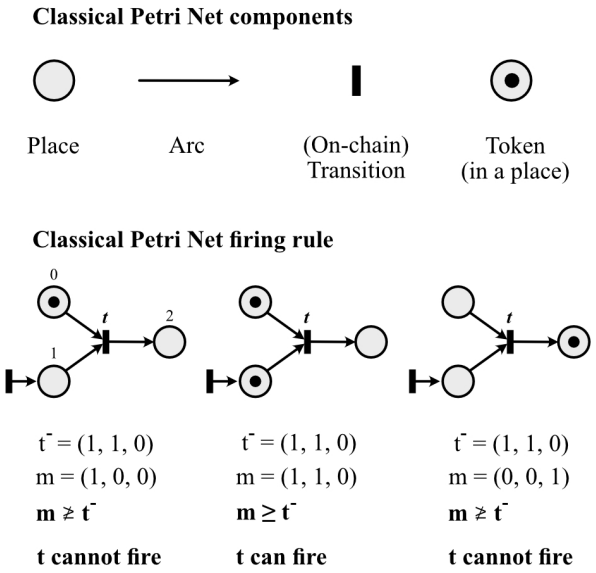

Figure 4: Classical Petri Net components consist of places, arcs, transitions and tokens. The classical Petri Net firing rule denotes that a transition can only fire if its input arcs are connected to places populated with tokens.

$$m \geq t_i^-$$

However, the Trace-Net skeleton (T,F,P) does not yet encode the possible satisfaction paths implied by each output script. Since a Bitcoin transaction output script can be satisfied with multiple satisfaction paths, we define each input arc to model the potential spending along a specific satisfaction path of an output. The possible spending along different satisfaction paths of a single output is modeled by different on-chain transitions in the Trace-Net model. It is possible they have the same transaction hash (txid) if they only differ in witness data.

By extension, a given place representing an output with multiple satisfaction paths should feature multiple input arcs from different on-chain transitions, each representing a single satisfaction path and respective witness encoding w. For a given on-chain transition t, the vector of all witness encodings assigned to its respective input arcs is given by large $W(t)$. The index in the set $W(t)$ represents the respective input index.

$$W(t) := \{w_0, w_1, ..., w_i\}$$

Since each input arc of an on-chain transition can only connect to a single place and be assigned to a single associated witness encoding, the following must hold for all places p and t in a Trace-Net.

$$| \, SatPaths(out(p)) \cap W(t) \, | \leq 1$$

An input arc only represents a single, unique satisfaction path in order to model the spending of different satisfaction paths with different on-chain transitions. A satisfaction path that only features public keys generated by a single actor implies a sweep transition, which moves the output funds to a destination determined by the single signing actor. This holds true because only signatures are the only satisfaction elements which can commit the outputs of a transaction. This notion is reflected in the creation of sweep transitions during Trace-Net model generation in subsequent section 2.5.1.

2.3.2 On-chain Contract Transition Markings

On-chain contract transition markings are applied to input arcs and transitions in the Trace-Net skeleton and express additional contract state governing an actor's ability to produce valid witness data. For an on-chain transition to be fire-able by a single or both actors, all markings must have released the respective on-chain transition. We define the following transition marking types.

- $h_{older}(t,i)$ - Input arc delay marking
- $h_{after}(t,i)$ - Input arc time marking
- $w_{actor}(t)$ - Transition witness marking

$h_{older}(t,i)$ markings exist for each input arc connected to a transition and reflects the number of confirmed blocks since the confirmation of the output it is spending. We provide a recursive definition of $h'_{older}(t,i)$, which is the delay marking of input arc (t,i) after a time delay of d:

$$h'_{older}(t,i) := \begin{cases} h_{older}(t,i) + d & iff \quad m \geq t_i^- \\ 0 & iff \quad m \ngeq t_i^- \end{cases}$$

Note that the $h_{older}(t,i)$ delay marking does not begin to increment before the connected place is populated with a token. For an input arc delay marking at index i to release its on-chain transition t, it must increment beyond an earliest firing time $eft_{older}(t,i)$, which in turn is determined by the presence of an older time-lock in the associated output satisfaction branch encoded by $W(t)[\,i\,]$ or $W(t)_i$. In the absence of an older time-lock, $eft_{older}(t,i)$ is set to 0.

The $h_{after}(t,i)$ input arc time marking simply holds the value of the chain height at the time of evaluation. It releases the on-chain transition at the earliest firing time which is determined by the after time-lock of the output its input arc is connected to. In absence of an after time-lock, $eft_{after}(t,i)$ is set to 0.

Finally, the $w_{actor}(t)$ transition witness marking describes which actor is capable of potentially firing this on-chain transition by independently producing the satisfying witness data without any additional communication between the actors. Since each input arc implies a single, specific witness encoding in the Trace-Net model, $w_{actor}(t)$ denotes which actors can satisfy all associated witness encodings of its input arcs for a on-chain transition t. We expand our definition of $CanSat(w,actor)$ to the entire set of assigned witness encodings $W(t)$ of a transition t:

$$CanSat(t,actor) := \begin{cases} 1 & iff \; CanSat(w,actor) \; for \; w \in W(t) \\ 0 & otherwise \end{cases}$$

We can now express the definition of the transition witness marking $w_{actor}(t)$ in the following form:

$$w_{actor}(t) := \begin{cases} int & iff \; CanSat(t,int) \wedge \neg \, CanSat(t,ext) \\ ext & iff \; \neg \, CanSat(t,int) \wedge CanSat(t,ext) \\ int,ext & iff \; CanSat(t,int) \wedge CanSat(t,ext) \\ \sharp & iff \; \neg \, CanSat(t,int) \wedge \neg \, CanSat(t,ext) \end{cases}$$

We have now defined all firing rules for on-chain transactions. A transition t can fire if its pre-places are populated, and all input arc and transition markings have released.

- $m \geq t_i^-$
- $h_{older}(t,i) \geq eft_{older}(t,i)$ for all input indices i
- $h_{after}(t,i) \geq eft_{after}(t,i)$ for all input indices i
- $w_{actor}(t) \neq \sharp$

If the above requirements are all satisfied, the actor(s) which can fire the transition t are given by $w_{actor}(t)$.

2.4 Trace-Net Contract State

The full contract state can now be described with the following tuple z:

$$z := (m, h_{older}, h_{after}, w_{actor})$$

We summarize the general relationship between the different contract transition types and contract state: The state of unspent outputs is described by place marking m, which changes when an on-chain transition fires. Both chain-state and time transitions affect the state of input arc markings h_{older}, h_{after}. Finally, the on-chain transition marking w_{actor} tells us which actors can currently produce the valid witness data for a given on-chain transition at a given contract state z. The state of w_{actor} is affected by the firing of off-chain transitions of type e. The firing of an on-chain transition always implies that the firing of off-chain transitions related to the script-locks it satisfies can no longer fire: These are revealed to the external observer during transaction broadcast.

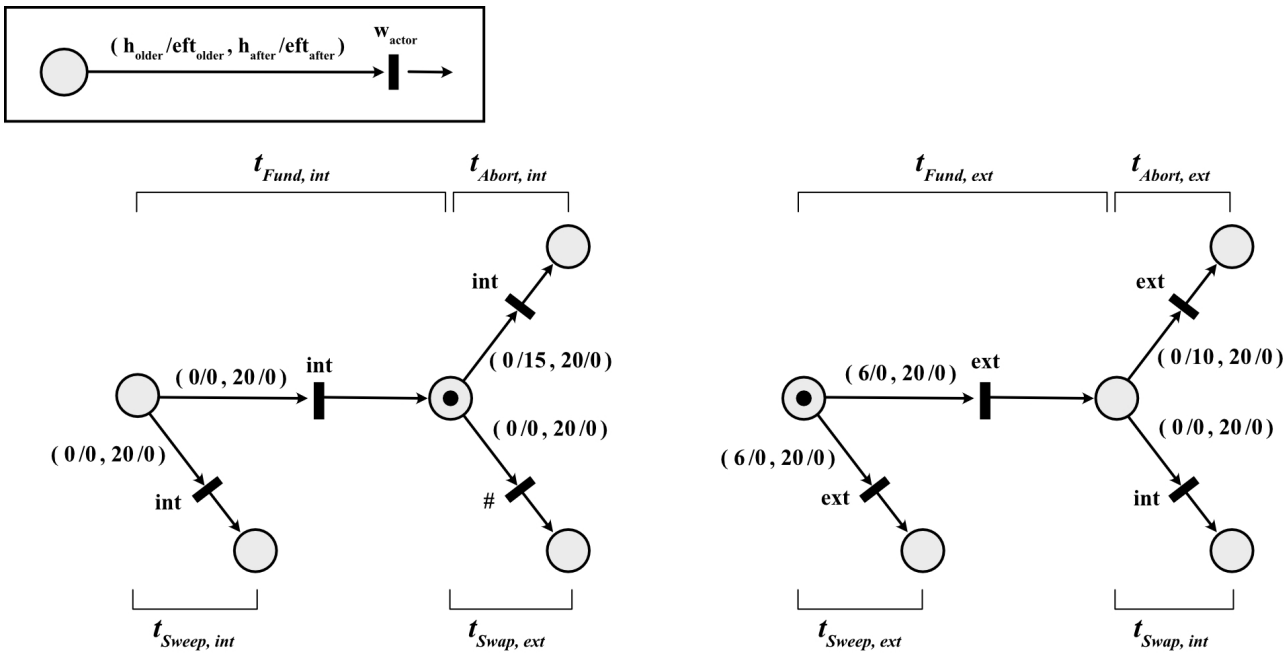

Figure 5: This illustrates a Trace-Net model of an atomic swap initiated by the internal actor. Input arc and transition markings are overlayed on top of the Trace-Net skeleton. The state shown above represents the contract state z imminently after the confirmation of the internal actor's funding transaction. Note the chain-height is 20 at time of evaluation, and that abort delays are set to 15 and 10 blocks respectively.

An example of a contract state is depicted in figure 2. This represents the contract state of an atomic swap initiated by the internal actor after the contract has been unilaterally funded by the verifying party.

2.5 Trace-Net Model Generation

We now detail how a Trace-Net contract model is generated from a set of Bitcoin contract transaction templates C and the individual satisfaction paths of $W(tx)$ associated with each transaction template input. The generation of a Trace-Net skeleton consisting of places P, transitions T and arcs F is described in algorithm 1.

Trace-Net model generation reflects the definitions and rules described in section 2.3.1. For each transaction template in contract C, a place and input arc are constructed for every input and its previous output (lines 6, 7 of algorithm 1). This place is generated with the amount and satisfaction paths of the previous output out_{in} the spending input is referencing. The input arc is associated with a witness encoding $W_{idx}(tx)$, representing the intended satisfaction path for the transition, which in turn implies its eft_{older} and eft_{after} values. $W(tx)$ encodes intended satisfaction paths for the transition input arcs and determines the script-locks under evaluation by transition marking $w_{actor}(tx)$.

Furthermore, an output arc and place is generated for each transaction template output amount. The place is associated with the amount and set of satisfaction paths (line 11 of algorithm 1) of the transaction output it is representing.

2.5.1 Sweep Transitions

The generation of a Trace-Net model in algorithm 1 also includes the generation of sweep transitions (lines 17-33), which are not explicitly represented in the contract transaction template set C. For each satisfaction path of each place modeled in a Trace-Net model which only features signature script-lock types generated by a single actor (lines 19, 25), we must model an additional, implied sweep transaction. This single-signer satisfaction path may include other script-locks, but since signatures are the only witness element type which endorse and commit all elements of the transaction, it is the signing actor who ultimately determines the destination of a single-signer transaction. In algorithm 1, $IntSatPath$ and $ExtSatPath$ are modelled to represent generic output satisfaction paths controlled by the internal and external actor respectively.

Algorithm 1 Trace-Net Model Generation

$P, T, F := \emptyset, \emptyset, \emptyset$

for all $tx \in C$ **do**
  $t := transition$
  Add t to T
  for all $in, idx \in tx$ **do**
    $p := place(amount(out_{in}), SatPaths(out_{in}))$
    $arc_{in} := arc(p,t)$ with $W_{idx}(tx)$
    $P := P \cup \{p\}$
    Add arc_{in} to F
  end for
  for all $out \in tx$ **do**
    $p := place(amount(out), SatPaths(out))$
    $arc_{out} := arc(t,p)$
    Add p, arc_{out} to P, F
  end for
end for

for all $p \in P$ **do**
  for all $w \in SatPath(p)$ **do**
    if all w signatures are internal **then**
      $t := transition$
      $p_{int} := place(amount(p), IntSatPath)$
      $arc_{in} := arc(p,t)$ with w
      $arc_{out} := arc(t, p_{int})$
      Add $t, p_{int}, arc_{in}, arc_{out}$ to T, P, F
    else if all w signatures are external **then**
      $t := transition$
      $p_{ext} := place(amount(p), ExtSatPath)$
      $arc_{in} := arc(p,t)$ with w
      $arc_{out} := arc(t, p_{ext})$
      Add $t, p_{ext}, arc_{in}, arc_{out}$ to T, P, F
    end if
  end for
end for

2.5.2 Initial Contract State

Determination of the initial contract state z_0 is implied by the on-chain state of outputs, the height of the block-chain and current ability of actors to satisfy the satisfaction paths for each input arc in the contract model at the time of contract verification. Figure 6 illustrates the partial state-space of an atomic swap contract initiated by the internal actor. The initial state at verification time in this example is imminently after the generation of contract transaction templates, before funding of the contract has occurred.

2.6 Contract State Space

Let $N = (P, T, F)$ be our Trace-Net skeleton and z_0 its initial state. We have now defined all transition types and their firing rules, so that at each state z, we can determine which on-chain, off-chain or time-transition can safely be executed. The reachability graph $RG(N, z_0) = (W, E)$ is a directed graph with the set of vertices W, and edges E. $RG(N, z_0)$ is generated with algorithm 2.

Algorithm 2 Contract State Space Generation

$W, E := \{z_0\}, \emptyset$

for all $z \in W$ **do**
  for all non-time θ fire-able in z **do**
    Compute z' such that $z \xrightarrow{\theta} z'$
    Add z' and (z, θ, z') to W, E
  end for
  for all on-chain t fire-able in z after min. delay d **do**
    Compute z' such that $z \xrightarrow{d} z'$
    Add z' and (z, d, z') to W, E
  end for
end for

Algorithm 2 exhaustively generates the contract state space reachable from an initial state z_0 by recursively computing all child states directly reachable via contract transitions fire-able in the parent state. The time delay transition is only fired in a state where no on-chain transitions are fire-able and an additional condition is true: A child-state with at least one fire-able transition must be reachable with a minimal time-transition d. The time-transition delay is chosen to be minimal, in order to reflect the transaction execution order enforced by contract time-locks. Figure 6 illustrates a contract state space example. It shows the partial reachability graph for an atomic swap contract initialized by the verifying, internal actor.

A feasible contract run σ will traverse the reachability graph RG(N) from z_0 until a terminal state z_n is reached, from which no further transitions can be fired.

$$\sigma = z_0 \xrightarrow{\theta_0} z_1 \ldots z_{n-1} \xrightarrow{\theta_{n-1}} z_n$$

Contract verification considers the set of feasible contract runs and determines whether they lead to a state which returns the correct internal balance and can be unilaterally executed by the internal actor.

3 Contract Safety

A terminally safe state z_s is one in which the marking m_s represents the expected confirmed balance amount spendable by the internal actor. In other words, the contract in state z_s is closed if the expected contract balance is spendable from outputs which exclusively feature $IntSatPath$ satisfaction paths.

The goal of contract state verification with Trace-Net is to exhaustively assess whether a contract in a given state

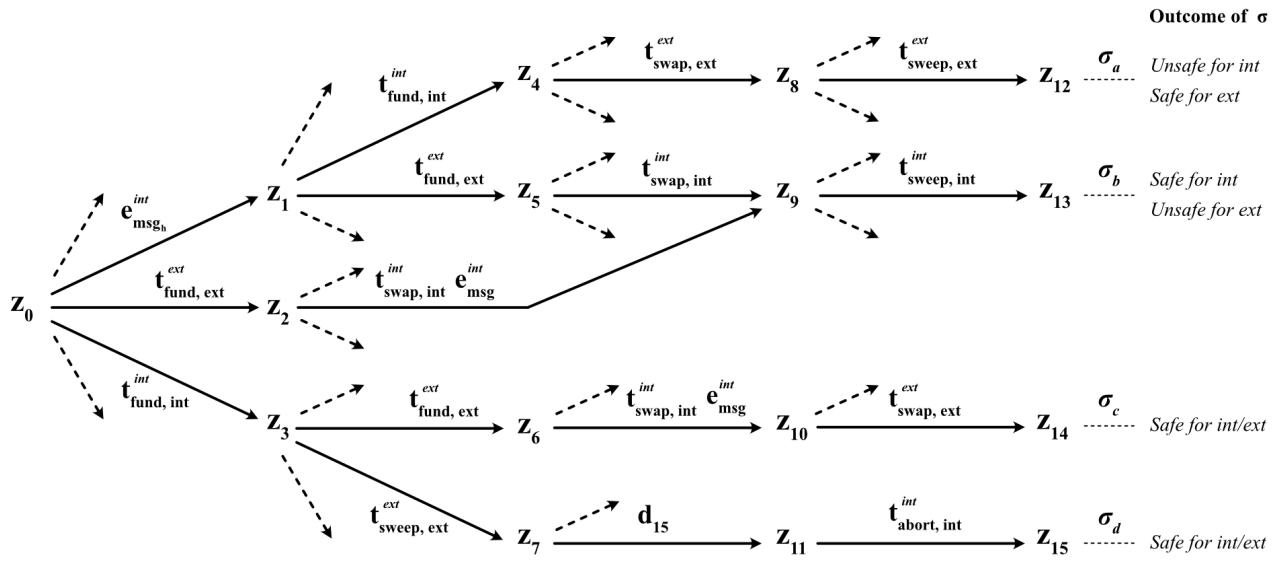

Figure 6: A partial reachability graph for our atomic-swap contract example is shown above. The contract is being evaluated for its initial state z_0, before any funding transaction has been executed. Each contract execution trace shown is denoted with the outcome of its execution.

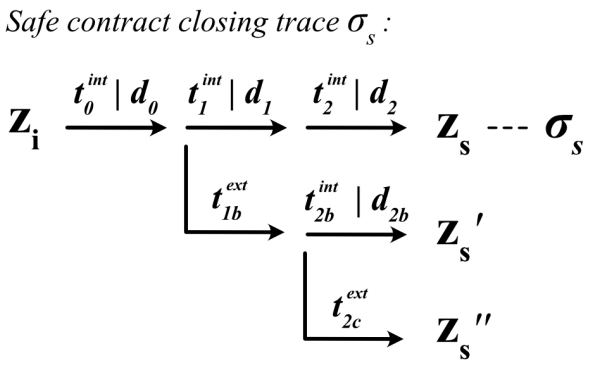

Figure 7: A safe contract exists at state z_i if an internal actor can unilaterally execute a path which leads to a terminally safe state (z_s, z_s', z_s''). This path must also be safe against competing transitions fired $(t_{1b}^{ext}, t_{2c}^{ext})$ by the external actor.

z_i can be safely closed by the internal, verifying actor. The safe contract closing trace σ_s beginning at state z_i must be unilaterally executable. Furthermore, such a contract closing trace must be robust against competing actions by the external actor. A safe contract closing trace σ_s fulfills the following criteria:

1. Reaches a terminally safe state z_s in $RG(N, z_0)$ from z_i.
2. Each transition in σ_s must be an internally fire-able on-chain transition t^{int} or delay d.
3. If trace σ_s includes an intermediary state z^* which features on-chain transitions t^{ext} fire-able by the external party, these alternative transitions must lead to states z^{**} in $RG(N, z_0)$ with at least one safe contract trace beginning at z^{**}.

States in criteria (3) can be interpreted as states for which there exists a race condition between transitions fire-able by different actors. In such a case, contract verification must ensure that a safe contract closing run can be executed from the subsequent states resulting from each of the race transitions. This ensures that the internal party can always execute a safe contract trace even if the unilateral run has been interrupted by an external party (Figure 7).

Throughout the life-cycle of a contract, actors may negotiate to collaboratively transition to different contract states.

The internal actor must verify that each new contract state is a safe state z_i with a safe contract closing trace σ_s.

3.1 Generalized Signing Policies

Trace-Net enables the definition of protocol-independent and generalized signing policies for a key management application which manages script-lock secrets used in a contract. Generalized signing policy dimensions which can be verified and enforced with Trace-Net verification include:

- Implied transfer of funds.
- Worst-case number of on-chain transitions required to withdraw funds from contract.
- Worst-case contract execution duration required to withdraw funds from contract.

Generalized signing policies may also include permitted levels of change to the criteria above during a contract update. For example, a contract update may increase the worst-case number of on-chain transactions required to close the contract, thereby increasing the projected transaction fee cost with the update.

Currently, key manager policies are implemented using contract transaction template matching, and are specific to a given contract protocol. Although this allows for secret keys to be managed separately from the contract template generating application, it has the disadvantage that contract types cannot be easily combined for better interoperability, efficiency or privacy. For example, it may be advantageous to combine a payment channel commitment state transaction with features from a coin-join transaction, in order to obfuscate coin ownership or payment-channel closing events. Enforcement of signing policies across non-standard contract types is only possible if the contract verification is conducted in a generalized manner, as we have demonstrated with Trace-Net.

We note that the introduction of a generalized signing policy implementations also introduces a novel design space for interactive protocols: Contract generation and secure signing can now be decoupled. A participating contract participant may not need to run a specific protocol implementation to be able to securely verify and sign contracts, but instead can verify any received contract proposal against an internal signing policy before endorsing the contract. This suggests the possibility of higher-order contracting protocols, in which general contract policies and their execution trace properties are negotiated between untrusting actors, rather than explicit contract transaction templates.

4 Conclusion

We have introduced a novel formal method to automate verification of multi-party Bitcoin contracts with transaction outputs featuring Miniscript. A key manager can automatically verify the relevant state space of a given Bitcoin contract or contract update, and ensure that the expected contract balance can be safely withdrawn by the contract verifier. This is a first step towards generalized key management systems which can safely consolidate signing functionality across different multi-party contracting protocols, enabling higher protocol interoperability and the potential to improve scalability and privacy.

Acknowledgments

Thanks to the reviewers of the early versions of this paper. A special acknowledgement to Josh Harvey for early discussions on formal Petri Net specifications of automata-based programs which inspired the idea of Trace-Net for Bitcoin contracts.

References

[1] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. https://lightning.network/lightning-network-paper.pdf, January 2016.

[2] Gregory Maxwell. Coinjoin: Bitcoin privacy for the real world. post on bitcoin forum. https://bitcointalk.org/index.php?topic=279249, August 2013.

[3] Rusty Russell. [lightning-dev] full disclosure: Cve-2019-12998 / cve-2019-12999 / cve-2019-13000. https://lists.linuxfoundation.org/pipermail/lightning-dev/2019-September/002174.html, September 2019.

[4] Pieter Wuille. Miniscript language website. http://bitcoin.sipa.be/miniscript, 2019.

[5] Pieter Wuille and Andrew Poelstra. Miniscript: Streamlined bitcoin scripting. https://medium.com/blockstream/miniscript-bitcoin-scripting-3aeff3853620, September 2019.

[6] Andrew Poelstra. Miniscript rust implementation. https://github.com/apoelstra/rust-miniscript, 2019.

[7] Pieter Wuille. Miniscript c++ implementation. https://github.com/sipa/miniscript, 2019.

[8] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *CCS '16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269, 2016.

[9] Sidney Amani, Myriam Begel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th*

ACM SIGPLAN International Conference on Certified Programs and Proofs, 2018.

[10] Yoichi Hirai. Formal verification of deed contract in ethereum name service. http://yoichihirai.com/deed.pdf, November 2016.

[11] Zeinab Nehaï, Pierre-Yves Piriou, and Frédéric Daumas. Model-checking of smart contracts. In *Proceedings of the IEEE International Conference on Blockchain*, 2018.

[12] Xiaomin Bai, Zijing Cheng, Zhangbo Duan, and Kai Hu. Formal modeling and verification of smart contracts. In *ICSCA 2018: 7th International Conference on Software and Computer Applications*, pages 322–326, 2018.

[13] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *FC'18 Proceedings of the 22nd International Conference on Financial Cryptography and Data Security*, 2018.

[14] Russell O'Connor. Simplicity: A new language for blockchains. In *PLAS '17: Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, September 2017.

[15] Louchka Popova-Zeugmann. *Time and Petri Nets*. Springer Verlag, 2013.

[16] Andrea Pinna, Roberto Tonelli, Matteo Orru, and Michele Marchesi. A petri nets model for blockchain analysis. *The Computer Journal*, 61:1374–1388, September 2018.

[17] Pieter Wuille, Jonas Nick, and Anthony Towns. Bip 341: Taproot: Segwit version 1 spending rules. https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki, January 2020.

[18] Pieter Wuille, Jonas Nick, and Anthony Towns. Bip 342: Validation of taproot scripts. https://github.com/bitcoin/bips/blob/master/bip-0342.mediawiki, January 2020.

[19] Pieter Wuille, Jonas Nick, and Tim Ruffing. Bip 340: Schnorr signatures for sec256k1. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki, January 2020.